AN ABSTRACT OF THE THESIS OF

Joel A. Oren for the degree of Master of Science in

Electrical and Computer Engineering presented on 8 February 1994 ,

Title: Design of an Asynchronous Third-Order Finite Impulse Response Filter

Abstract approved: Redacted for privacy

Sayfe Kiaei

With the increased demand for complex digital signal processing systems, real-time signal processing requires higher throughput systems. In the past, the throughput has been increased by increasing the clock rates, but synchronization can become increasingly more difficult. Recently there has been renewed interest in designing asynchronous digital systems. In an asynchronous system, there is no global clock, and all modules communicate through handshaking. In this thesis we demonstrate an implementation of an FIR filter using asynchronous digital circuit techniques. These asynchronous design techniques are used to test whether a practical signal processing filter can be implemented with asynchronous logic. A third-order four-bit filter is developed and simulated with SPICE, comparing favorably with other available technologies in speed and power consumption. Although in practice 8-16 bits are needed, this work is sufficient to demonstrate the feasibility of asynchronous circuits for filtering applications. A chip is laid out in 2 micron CMOS, and testing shows that it has a speed-power product comparable with asynchronous designs fabricated by others.

Design of an Asynchronous Third-Order Finite Impulse Response Filter

by

Joel A. Oren

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed February 8, 1994

Commencement June 1994

APPROVED:

Redacted for privacy

Assistant Professor of Electrical and Computer Engineering in charge of major

Redacted for privacy

Head of department of Electrical and Computer Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented February 8, 1994

Typed by researcher for Joel A. Oren

TABLE OF CONTENTS

1. INTRODUCTION 1
    1.1 FIR filter design with ECDL asynchronous circuits 1
    1.2 Historical Background 3
    1.3 Organization of this Thesis 3

2. DESIGN OF THE FIR FILTER 4
    2.1 FIR Filter description 4
    2.2 Operation of the FIR filter in a pipeline 5
    2.3 Data-Flow Interpretation of the Architecture 8
    2.4 Sequence of operation of the Data Flow filter stage 9
    2.5 Initialization of the Pipeline 12

3. ENABLE/DISABLE CMOS DIFFERENTIAL LOGIC (ECDL] 14
    3.1 Micropipelines 14
    3.2 The basic ECDL circuit 19
    3.3 Initial Token Generation 21

4. CIRCUIT DESIGN, SIMULATION, AND LAYOUT 25
    4.1 Overall organization of the Processing Elements 25
    4.2 The Control Unit 25
        4.2.1 The Muller C-element 25
        4.2.2 The XNOR 28
        4.2.3 The Flip-Flops 28
        4.2.4 Simulations 33
    4.3 Data Path 35
        4.3.1 Multiplier 36
        4.3.2 Adder 37
        4.3.3 Simulations 39
    4.4 Overall Structure of the Filter 43
        4.4.1 Floor plan of the Filter stage 43
        4.4.2 Layout issues for the filter stage 46
        4.4.3 Simulation results for the filter stage 49

5. TESTING AND EXPERIMENTAL RESULTS 53
    5.1 Measurement of Speed and Delay 53
    5.2 Test Results 58

6. CONCLUSION 61

BIBLIOGRAPHY 63

LIST OF FIGURES

Figure Page

1. General 3-stage filter 5
2. Four steps in an FIR filter operation 7
3. A non-recursive convolution filter 8
4. Data-Flow Architecture of the Filter 9
5. Four steps in the operation of the Data-Flow filter stage 10
6. Sutherland's Micropipelines 16
7. Timing Diagram for the Micropipeline. 16
8. Modified ECDL Micropipeline. 17
9. Basic ECDL circuit 19
10. Timing Diagram for the Micropipeline. 21
11. Two-stage ECDL pipeline. 23
12. Muller C-Element logic symbol, circuit diagram, and state diagram. 26
13. Basic XNOR circuit 28
14. Double-Edge-Triggered D Flip-Flop Diagram 29
15. SETDFF (Single-Edge-Triggered D Flip-Flop). 32
16. Multiplier Block Diagram. 36
17. Adder Block Diagram. 38
18. Multiplier Simulation Results. 40
19. Adder Simulation Inputs 41
20. Adder Simulation Results. 42
21. Floor diagram of the Triplet filter stage. 44
22. Plot of the filter chip. 45
23. Triplet Simulation Results. 49

24. Triplet Simulation Results.....50
25. The critical path through the last stage.....56
26. State analyzer graph for the filter chip.....59
27. Oscilloscope plot for the filter chip.....60

LIST OF TABLES

Table Page

1. Logic Table for the Muller C-element with 1 inverted input. 27
2. State Transition Table for the Muller C-element with one inverter, ...27
3. State Transition Table for the XNOR circuit. 28
4. State Transition Table for the DETDFF Latch 1 circuit. 30
5. Summary of simulation results for the Control-Path circuits. 35
6. Signal names for the Multiplier simulation results. 39
7. Signal names for the Adder simulation results. 43
8. Summary of simulation results for the Data-Path circuits. 43
9. Summary for the Triplet cell. 46
10. Signal names for the Triplet simulation results. 51
11. Comparison of 2u and 1.2u feature sizes for the Triplet filter cell. 52
12. Comparison of 4- and 8-bit Triplet filter cells. 52
13. Maximum cycle time for the filter stage 57

DESIGN OF AN ASYNCHRONOUS THIRD-ORDER FINITE IMPULSE RESPONSE FILTER

1. INTRODUCTION

1.1 FIR filter design with ECDL asynchronous circuits

Finite Impulse Response (FIR) filters are increasingly found in applications such as Delta-Sigma modulators as decimation filters to reduce the noise. An FIR filter is a filter that has a finite response to a finite input. The transfer function of a typical FIR filter could be shown as:

$$h(u) = \sum_{i=1}^{M} a_i * x(n-i) \qquad (1.1)$$

The advantages of FIR filters are that they have constant phase and group delay, and are stable. Disadvantages include the roundoff noise associated with all digital filters, the need for a large number of stages to get sharp cutoff characteristics, and complex design compared to IIR (Infinite Impulse Response) filters [RO86]. As the speed requirements for these filters increase, the difficulty in realizing a synchronously clocked chip increases due to area, resistance, and clock skew effects [SU89] [WE88]. In response to such clocking difficulties, asynchronous or self-timed circuits are being investigated.

This thesis presents a self-timed design for a four-bit third-order FIR filter using the two-cycle micropipeline for control [LU88]. We will compare this design with other available circuits and filter architectures, and describe its advantages and disadvantages in terms of power consumption, speed, and area. The filter designed in this thesis has been fabricated in a 2 micron CMOS process, and both simulated and measured results will be presented.

Asynchronous systems excel over synchronous systems in that they can proceed with processing upon the arrival of the input signals. Furthermore since there is no global clock, the control of the system is easier and the delays are substantially lower. In asynchronous circuits, handshaking is used to pass values from one stage to the next in a decentralized fashion. This allows the calculations within the FIR filter to proceed as fast as possible. As larger filters are constructed with longer latencies and more complex layouts, the use of asynchronous circuits will give greater performance improvements than comparable filters implemented in standard synchronous logic. This is because the asynchronous filters will not have the time delays between cycles that must be built into synchronously clocked filters.

Another method to synthesize self-timed circuits is by using data-flow diagrams. Asynchronous circuits are data-flow machines where each data has an associated token. The computation flows through the data flow machine which consists of nodes. These nodes "fire" or perform their logic function whenever all the tokens and associated data are available. The goal of the FIR filter design presented here is to use the asynchronous architecture to speed up calculations by allowing the availability of data to determine how fast the computation proceeds.

In order to fully exploit the advantages of the self-timed architecture, the logic family used must be able to perform handshaking and generate a completion signal. The circuit type used for the filter described in this thesis is based on Enable/Disable CMOS Differential Logic (ECDL) [LU91A]. ECDL is a differential logic family which can generate both signals and their complements. ECDL is based on the static CMOS logic family and it has the CMOS advantages of lower static power, higher noise margins, and easier implementation.

1.2 Historical Background

There have been several recent works for FIR filter design using systolic arrays for the processing of signals [QU91]. Using pipelining, fast processors can achieve orders of magnitude higher throughput. Prior work in asynchronous signal processing circuits was based on Differential Cascade Voltage Swing Logic (DCVSL). Practical filters were designed using DCVSL [JA88]. The DCVSL-based filters used Sutherland's four-cycle micropipelines for control [SU89]. This work will apply the two-cycle technique to implement the system.

1.3 Organization of this Thesis

The organization of this thesis is as follows: Chapter 2 will describe the overall architecture, demonstrate the operation of the filter, and discuss the data-flow architecture and pipelining of FIR filters. Chapter 3 will discuss the ECDL circuit, the micropipeline, and initial token generation in the ECDL micropipeline. The fourth chapter will present the simulation results of the filter building blocks. A summary of the results comparing delay times and power consumption with other technologies will be given. Chapter 5 discusses the testing and experimental measurement from the fabricated IC. Finally, the sixth chapter will summarize the advantages and disadvantages of this design and give some ideas for future improvements.

2. DESIGN OF THE FIR FILTER

In this chapter we will discuss the architecture of the general purpose FIR filter. We will begin with an overview of the architecture at the block level and the algorithmic level. Next we will show the operation at the block level with a series of examples. The third part of this chapter discusses the data-flow architecture of this chip. Operation of the circuit is demonstrated by a series of "snapshots" of the filter. The last section introduces the topic of initial token generation.

2.1 FIR Filter description

Figure 1 shows the pipeline diagram of a third order FIR filter. The third order FIR filter realizes equation 2.1 below.

$$Y_{out}(n) = Y_{in}(n) + a_1 * x(n) + a_2 * x(n-1) + a_3 * x(n-2) \qquad (2.1)$$

where a1-a3 are the coefficients of the filter, the x values are the data inputs to the system, and the Y values are the results flowing through the system. The FIR filter has advantages compared to IIR (Infinite Impulse Response) filters. It has a linear phase response and is easier to design than an IIR filter.

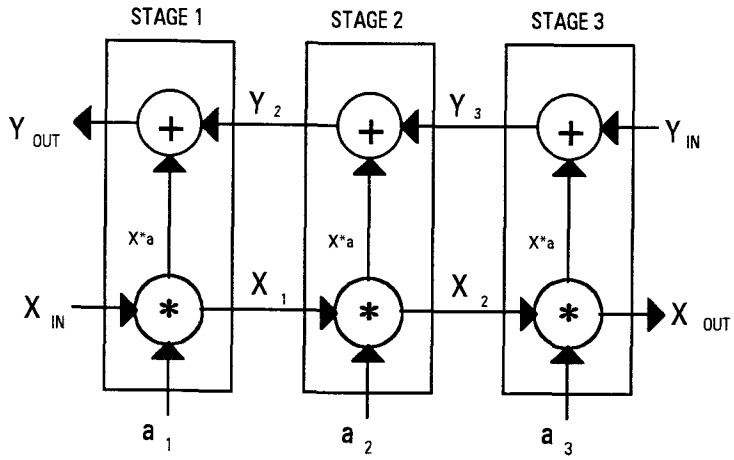

Figure 1. General 3-stage filter

In general, each block of this filter performs the following operation:

$$Y_{out} := Y_{in} + a * X_{in} \qquad (2.2)$$

Using these basic modular blocks, filters of higher order can be implemented. Each stage of the filter requires one adder and one multiplier and for the Nth order pipelined filter, N adders and N multipliers are required.

A direct implementation of the FIR filter uses only one adder and one multiplier to compute the result Y_{out}. It could do the computation by computing all sums and products on one modular block, by multiplexing the filter block in time. This would result in a reduction of the computation rate. The FIR filter is a non-recursive convolution network. The non-recursive aspect of the filter gives it it's finite output response. The block diagram of a fourth order, four-bit FIR filter in a pipelined (systolic) [QU91] fashion is shown in Figure 3.

2.2 Operation of the FIR filter in a pipeline

This section examines the operation of the pipelined filter in more detail. Figure 2 shows snapshots of the array of filter stages. The data stream x values

enter from the left to the right without being modified. We are assuming that there is a host I/O interface. The other stream is the result y stream, which flows from right to left. The initial value of Y_{in} is either from a preceding module or else it is zero.

In part (a) of Figure 2, the token Y_i is being processed, along with X_i, in the first stage of the pipeline. The tokens Y_{i+2} and X_{i-2} have just been presented. In part (b) the token Y_i is output, and the next input token X_{i+1} is being presented. Parts (c) and (d) of Figure 2 trace another cycle of operation of this pipeline.

It should be pointed out that in part (a) the first and third stages are processing and the second and fourth stages are idle. The opposite is true in part (b), which means that each stage is only operating half of the time. The ECDL micropipeline filter design presented here will remove this inefficiency.

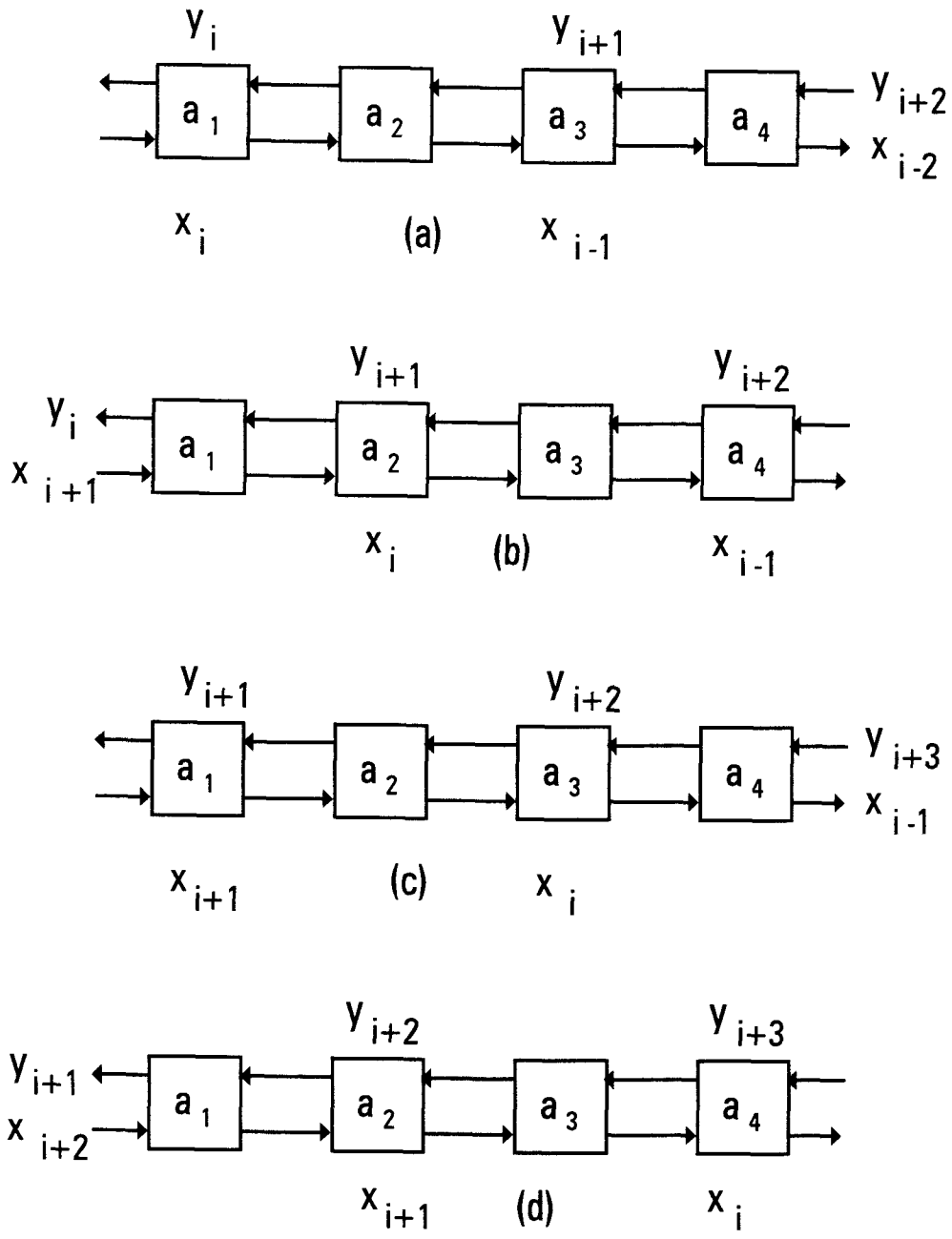

Figure 2. Four steps in an FIR filter operation

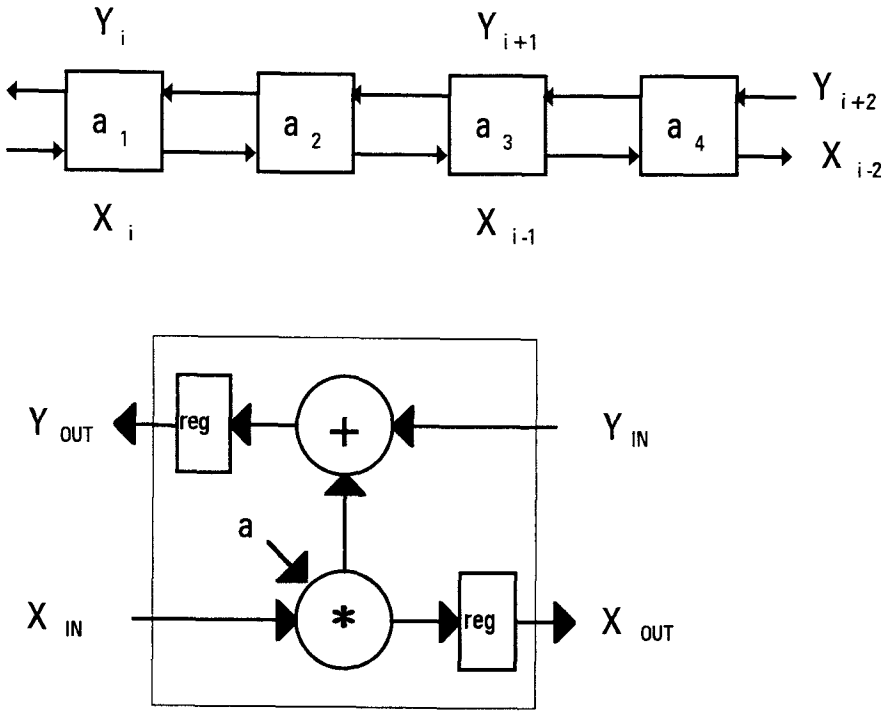

Figure 3. A non-recursive convolution filter

Figure 3 shows one of the PE blocks in more detail. Since the circuit will be asynchronous, storage will be necessary to hold the tokens while the pipeline has stopped. The input token X_{in} will flow through the multiplier, where it will be multiplied by the coefficient a, and then will be used by the adder. The input token Y_{in} is flowing into the adder, and may either come from a subsequent stage or it may be a zero (for the last stage).

2.3 Data-Flow interpretation of the Architecture

The FIR filter can also be described using the data-flow graph [AR86] [LA88]. The data flow graph contains nodes for adders, multipliers, forks, and joins. There is a fork required at the beginning of each input stage. This fork is required to split the input X token to be used by both the multiplier and the next stage. The coefficients are assumed to be constant, and so the multiplier has

only one asynchronous input, for the X_{in} token. The join is required to synchronize the result token of the multiplication with the Y_{in} token generated by the subsequent stage. An adder generates the Y_{out} token to pass to the next stage, or in the case of the last stage, to the outside of the chip.

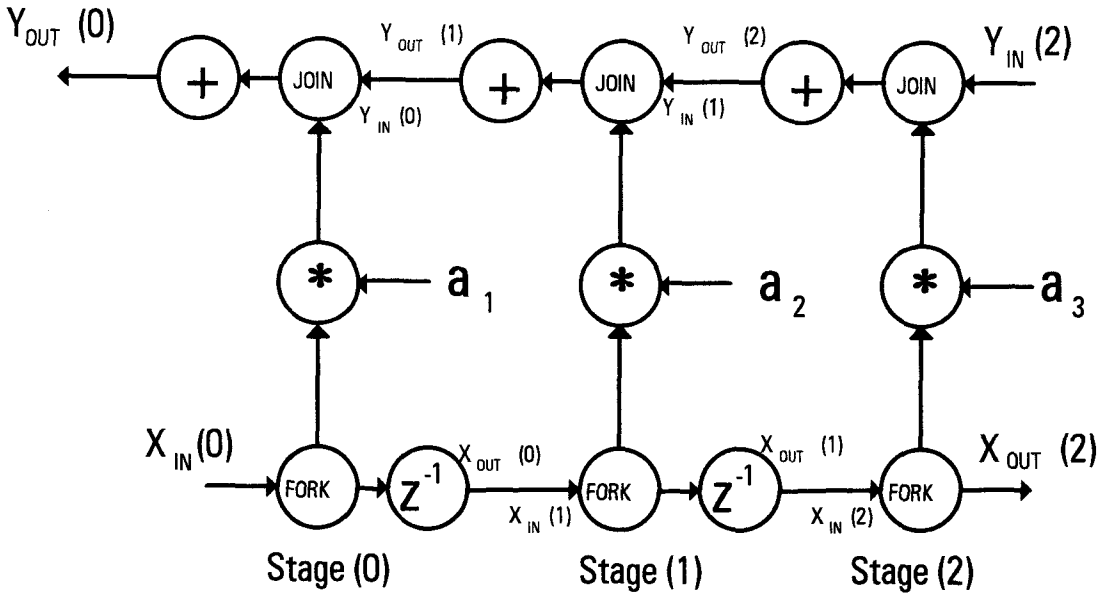

Figure 4. Data-Flow Architecture of the Filter

2.4 Sequence of operation of the Data Flow filter stage

Similar to the previous snapshots, we will look at the operation of the data flow pipeline in greater detail. In Figure 5, the operation of one stage of the data-flow filter is examined. As with the FIR block diagram in Figure 3, the X data values are flowing from left to right, and the Y result values are flowing from right to left. In this case, the Y_{in} values are usually non zero, except in the case of the last stage in the filter. The X data values are not modified, but are delayed one cycle.

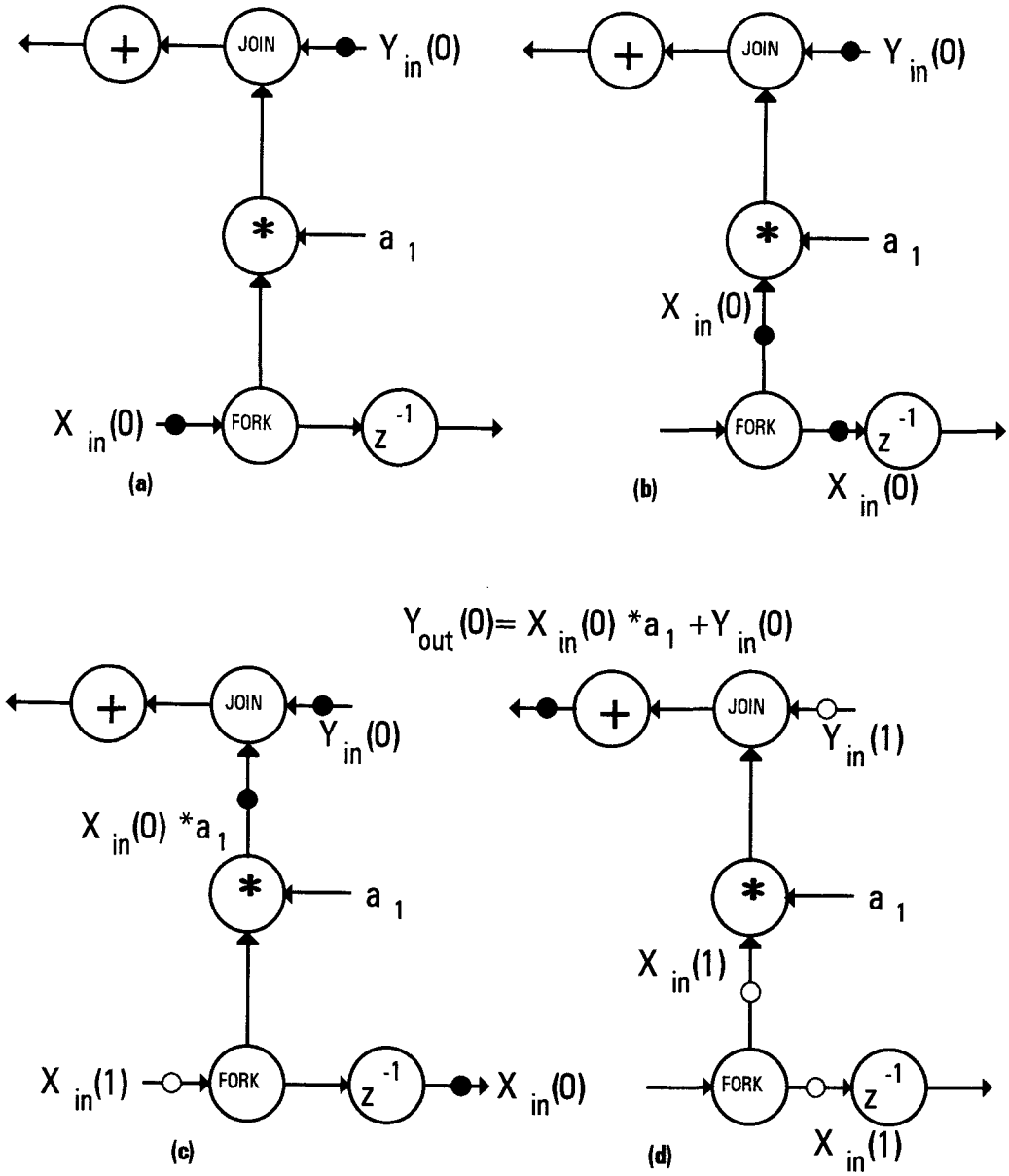

Figure 5. Four steps in the operation of the Data-Flow filter stage

Figure 5(a) shows X_{in} data and Y_{in} result tokens (filled) waiting at the input to the filter stage. These tokens do not necessarily have to appear simultaneously. Figure 5(b) shows the X data token going through the fork. The fork produces two X data tokens, which go to the delay and the multiplier. In Figure 5(c) the X token goes through the delay, and then the multiplier produces a product token for X times the coefficient a_1. The coefficient a_1 is a constant

and is not set up as a token. The product token $X_{in}(1)$ and the $Y_{in}(1)$ input token are now waiting at the join, which is enabled as shown in Figure 5(c). In the Figure 5(d), the join has fired and passed the two matched data tokens on to the adder, which has produced the final output Y token.

If there are no tokens in the pipeline when the X_{in} data tokens arrive, the circuit will not operate properly. For example, consider a single X_{in} token presented to the pipeline in Figure 4. As X_{in} enters the first stage, it is split and goes on to the current stage multiplier and to the next stage. In the first stage, the multiplication result a_1*X_{in} will be waiting for a Y_{in} token to arrive before the adder can fire. The same thing could happen in the next stage, resulting in three multiplication values waiting at the multiplier inputs to the joins. In this case, when the $Y_{in}(0)$ token is presented, it triggers the last stages' join, which fires and enables the last stages' adder. The $Y_{in}(0)$ token plus the result of the multiplication in stage 2, $a_3*X_{in}(0)$, are added and sent to stage 1. However, there is already a token, $a_2*X_{in}(0)$, waiting at the input to the join in stage 1. The join fires, enabling the adder, which adds $Y_{in}(0)+a_3*X_{in}(0)$ to $a_2*X_{in}(0)$. A similar operation takes place in stage 0, resulting in:

$$Y_{out}(0) = Y_{in}(0) + a_3*X_{in}(0) + a_2*X_{in}(0) + a_1*X_{in}(0). \quad (2.3)$$

The above is not the correct output. The Y_{out} token should depend only on the last three values of X_{in} and the Y_{in} token from three cycles before. However, in the case described above, the $Y_{out}(0)$ token depends only on the current value of Y_{in} and X_{in} tokens. The problem is that the first Y_{in} token presented immediately flows through all of the stages, instead of firing the last stage only. The solution is to place initial tokens in the pipeline as space holders, to set up the pipeline properly. The technique for generating initial tokens is the subject of the next section.

2.5 Initialization of the Pipeline

In order for the pipeline to operate correctly, a sequence of initial tokens must be automatically generated each time the pipelined circuit is activated. These initial dummy tokens, generated by the pipeline in the initial reset, will keep the filter from running through an entire sequence of operations on the first data token presented.

This problem is solved by inverting some of the tokens in the Y_{out} result stream. It was found through simulations that inverting the tokens in alternate stages will generate the initial tokens needed without interfering with the normal operation of the pipeline. This solution has the added benefit that it does not cause excessive slowing of the pipeline. Alternate token inversion causes the pipeline to pause for one cycle and wait for the next data to be presented before continuing. This is especially important at the initialization of the pipeline.

To demonstrate the initialization, we will refer to Figures 4 and 5.

1. As the first data value $X_{in}(0)$ enters the first stage, it goes through the fork in Processing Element 0 (PE0) and is split into two tokens, as demonstrated in Figure 4 (b).
2. The first token goes through the multiplier in PE0 to the join. There is a $Y_{in}(0)$ token waiting at the join, and the token from the multiplier is added to the $Y_{in}(0)$ token, and the $Y_{out}(0)$ result token flows out. The value of the $Y_{out}(0)$ token depends only on the value of the $X_{in}(0)$ and $Y_{in}(0)$ tokens.
3. The second $X_{in}(0)$ token generated by the fork in PE0 goes on to the fork in PE1, and is again split into two. One of these two X_{in} tokens goes on to PE2 and the other token goes through the multiplier in PE1 to the join.
4. There is a $Y_{in}(1)$ token waiting at the adder in PE1, generated by the initialization, so the $(X_{in}*a_2)$ token continues through the adder in PE1 resulting in a $Y_{out} = ((X_{in} * a_2) + Y_{in})$ result token.

5. The result token from the adder in PE1 flows back to PE0, where it is now a token waiting at the Y_{in} input to the adder in PE0. The $X_{in}(0)$ token in PE0 was already used and the multiplier in PE0 has already fired, so the Y_{in} token in PE0 must wait for the next X_{in} token to be presented to the pipeline. It is the first Y token waiting in the pipeline.

6. The first token generated in step 3 above, by the fork in PE1, flows straight through the multiplier in PE2 since there is no fork in PE2. It waits temporarily at the join in PE2 for the $Y_{in}(2)$ token.

7. When the $Y_{in}(2)$ token is input from an external source, the join in PE2 fires and triggers the adder in PE2. The adder result token $Y_{out} = Y_{in}(2) + (X_{in}(0) * a_3)$ is sent back to PE1 as the $Y_{in}(2)$ input token to the adder/join in PE1.

8. At the join in PE1 the result token from PE2 must wait, because it is the only token waiting at the join in PE1. There are now two Y tokens waiting in the pipeline.

To summarize this process, the result of inputting one X_{in} token and one Y_{in} token is one Y_{out} token and two partial result Y_{in} tokens waiting in the pipeline. The pipeline ends up the same as it was when it was initialized. It has two tokens waiting, one each at the joins in PE0 and PE1. It is now set up for the next pair of inputs.

Before we explain how the initial token generation is done, the ECDL circuit used is described. The ECDL circuit is at the heart of the modified micropipeline, and it's characteristics make initial token generation possible for this filter. After we have covered the ECDL circuit and the micropipeline in the next chapter, we will revisit the problem of initial token generation.

3. ENABLE/DISABLE CMOS DIFFERENTIAL LOGIC (ECDL).

The ECDL logic family was proposed by [LU88A, 88B]. It is a static logic family, but it differs from previous static differential logic families in that it dissipates very little static power. Differential logic families require that both an input and it's complement be presented, and generate both the output and it's complement. The ECDL family uses a single-phase clock, which reduces problems caused by clock skews.

Some previous designs of asynchronous logic systems have been based on Differential Cascade Voltage Switch Logic, or DCVSL [JA88]. The DCVSL family dissipates more static power and uses more devices than comparable circuits implemented in either static CMOS or ECDL. ECDL was developed to reduce the static power consumption and improve the speed/power product in asynchronous circuits.

In this chapter, we will first examine the micropipeline control structure designed by Sutherland [SU89], and it's modifications to take advantage of the special characteristics of the ECDL circuit. Application of ECDL to micropipeline control and the initial token generation for the FIR filter are then examined.

3.1 Micropipelines

The basic asynchronous circuit consists of the logic for the signal calculation and the controller called the micropipeline. The micropipeline is an asynchronous self-timed pipeline developed by Sutherland [SU89], which contain a string of stages, each stage containing computation and control logic. The control logic, shown in Figure 6, is composed of a Muller C-element, storage

for all variables, and a delay element. The Muller C-element synchronizes the current stage to the previous and next stage. A variable storage is needed since the previous stage can change it's outputs as soon as the current stage has acknowledged receiving them, and a delay element is required to guarantee that a request is not generated to the next stage before the current stage has finished it's computation. The micropipeline used in this work was modified to take advantage of the completion signal generated by the ECDL circuit [LU88A].

The token storage show in Figure 6 is necessary in order to store tokens between processing stages, so that the stage generating the token can proceed with its processing of new tokens. The token storage in the modified ECDL micropipeline should be contrasted with the token storage used in Sutherland's micropipeline. A double-edge-triggered flip-flop is used as the variable storage in Lu's micropipeline because of the two-cycle nature of the micropipeline control. The handshaking signals for Sutherlands storage register are as follows: The Capture input, C, signals the availability of the variable from the previous stage; The Capture Done signal, Cd, signals when this variable has been stored in the register (captured); Similarly, Pass (P) and PassDone (Pd) signal when the previously captured data is output to the current stage. To simplify this, the double-edge-triggered D-flip-flop is used in the modified ECDL pipeline. The remaining signals, R and A , are Request and Acknowledge handshaking signals between stages.

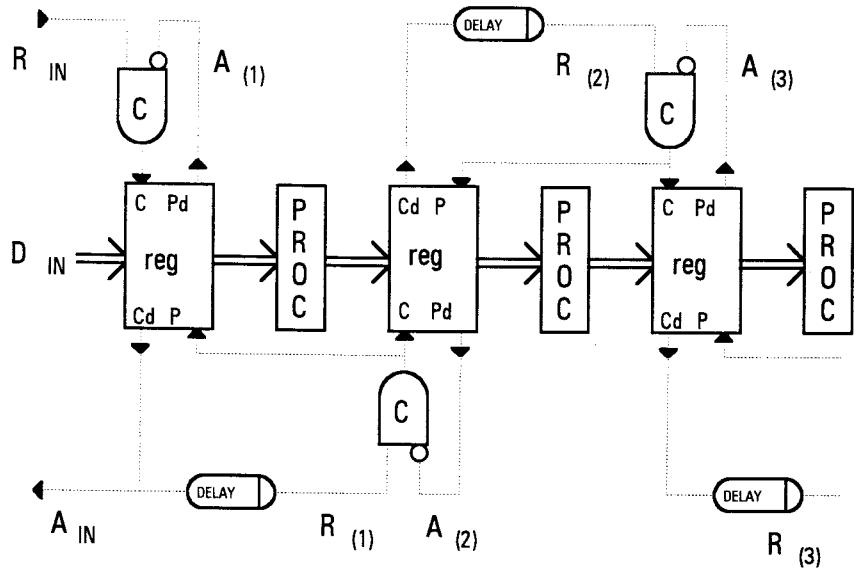

Figure 6. Sutherland's Micropipelines

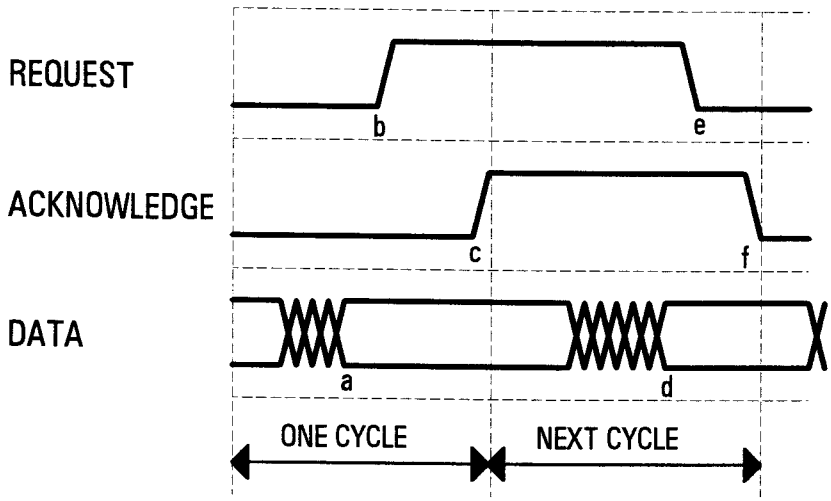

Figure 7. Timing Diagram for the Micropipeline.

The timing diagram in Figure 7 shows the sequence of events in the micropipeline: (1) At time a, the input data becomes valid for the first stage. The first stage has completed it's function and the $data_{out}$ signal lines are stable. (2) At time b the $Request_{out}$ is generated by the first stage and is received by the second stage. (3) At time c, the second stage has entered the

data values into a register, and signals the first stage via the Acknowledge$_{in}$ line that it no longer needs the data.

At this point, the next cycle starts, and the first stage can begin to compute the next data value. The sequence continues similarly to the first cycle except that the Request and Acknowledge signal lines have the opposite polarity from the first cycle. This does not affect circuit operation, since positive and negative signal transitions are equivalent.

Figure 8 shows the modified ECDL version of the micropipeline. This version has three differences compared to the micropipeline discussed above. First, the delay element is eliminated since ECDL can generate a completion signal with the addition of a NOR gate. Also, the register REG in Figure 6 for the input signal has been replaced with a Double-Edge-Triggered D flip-flop (DET in Figure 8) [LU90] [LU88A]. The clock signal for the SETDFF register generating REQout flows through the DETDFF input token storage register, unlike the Sutherland micropipeline where separate CaptureDone and PassDone signals controlled the input token storage register operation.

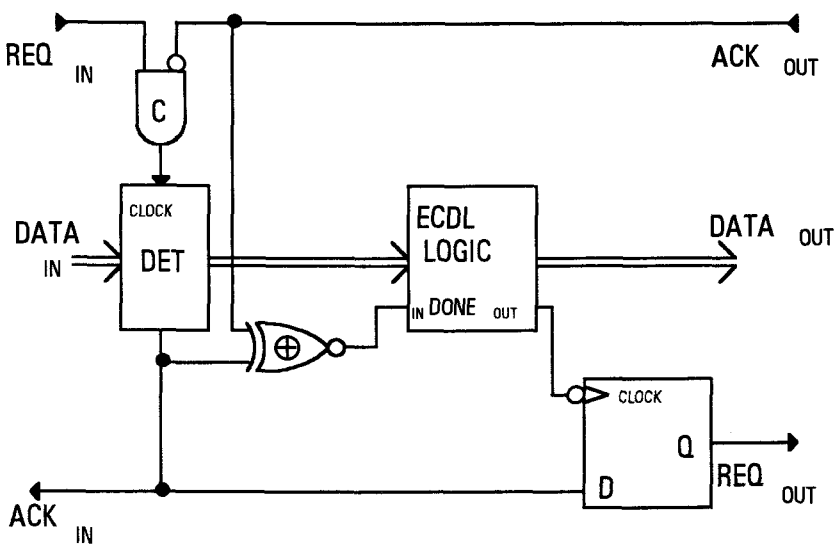

Figure 8. Modified ECDL Micropipeline.

ECDL is inherently a four-phase circuit and is level-sensitive rather than edge-sensitive. In a four-phase system, the signals are level sensitive and must make a full transition from low to high to low. Two-phase circuits are triggered by both the rising and falling edge, which consumes less energy and time.

The micropipeline is a two-phase circuit, therefore the modified ECDL micropipeline will require a circuit to convert between two- and four-phase and back again, which adds an XNOR and a SETDFF (Single Edge Triggered D Flip-Flop). The XNOR circuit is used to convert from two-phase to four phase, and the SET D flip-flop is used to convert from four phase back to two phase.

There are two Acknowledge lines in each stage of the micropipeline, one going to the previous stage and one coming from the next stage. They will never change at the same time, because the circuit will not accept new data from the previous stage until after the next stage has accepted the current data outputs. By connecting the XNOR gate to both the Request and Acknowledge lines, the XNOR will produce two transitions for each two-phase control cycle.

The SETDFF converts the four-phase output of the ECDL gate back into two-phase control logic by connecting the clock input of the SETDFF to the output of the ECDL gate to trigger the flip-flop once for every complete cycle of the ECDL circuit. Connecting the D input of the flip-flop to the $Acknowledge_{in}$ line (which is already two-phase) will generate one transition for every two transitions of the ECDL gate.

Another way of looking at the Micropipeline is to see it as a sequence of unstable loops. There is exactly one inversion in each loop around a stage in the Micropipeline, and we know that a signal path with an odd number of inverters will oscillate. The Muller C-elements and SETDFFs act to control the oscillation based on the state of the ECDL circuit.

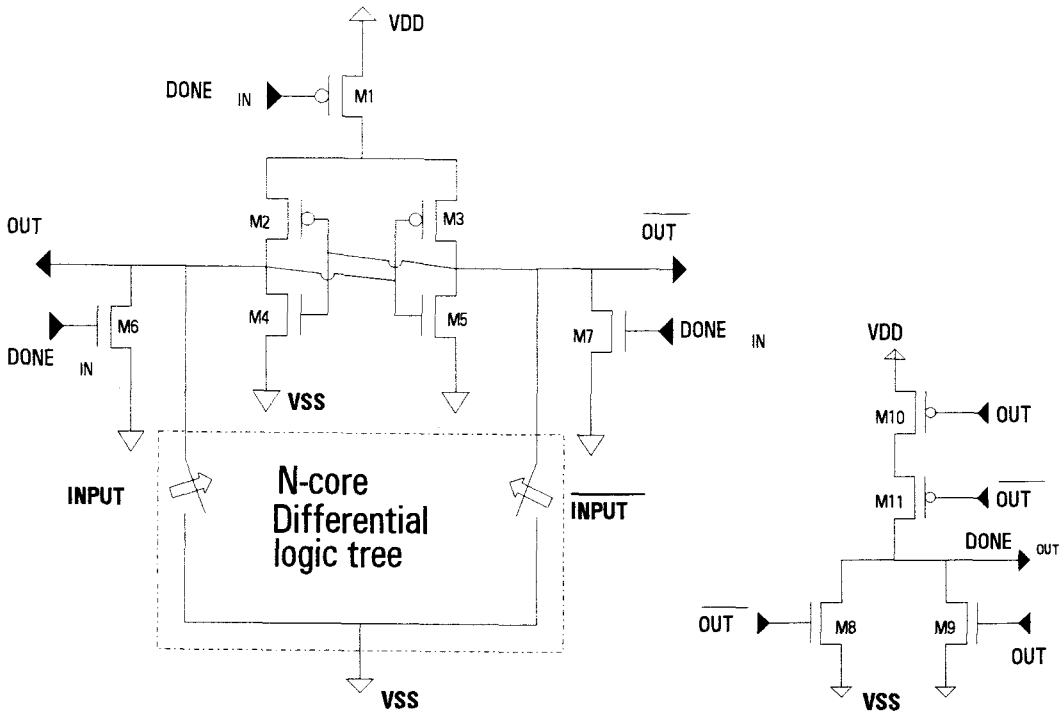

Figure 9. Basic ECDL circuit

3.2 The basic ECDL circuit

The ECDL cell shown in Figure 9 has several basic elements. In the center of the cell is the memory element, composed of the cross-coupled inverters M2-M5. Three transistors are used for control: M1 enables the circuit by connecting it to VDD; M6-M7 reset the circuit by connecting the outputs to VSS. The Differential logic tree generated the output by allowing one output to go to VSS and holding the other output high, depending on the input signals. Lastly, the NOR circuit generates a completion signal when the output values are valid.

When the data is available from the previous stage, the $Done_{in}$ signal goes low and activates the cell. The flip-flop value is set by the differential input tree structure connected to the outputs of the flip-flop. When the $Done_{in}$ signal

is high the cell is disabled. Devices M6 and M7 are enabled, and both outputs are low. The NOR circuit has both inputs low, and it's output is high, presenting a disable signal to the next stage.

When the $Done_{in}$ input signal goes low the circuit is enabled (event a in Fig. 8 below). Depending on the input signals, the differential tree holds one output at ground while the other output is left open. The output left open is pulled high by the action of one of the cross-coupled inverters in the memory element.

Since the circuit had the low voltage on the inputs to both of the inverters, both of the inverters will be trying to bring their output positive. Only one will be successful, the other being overridden by the differential tree structure. The remaining two transistors are used to drag the outputs of the flip-flop (the inputs of the two inverters) low during the inactive phase of the circuit. The delay time required for the logic tree and the inverters to bring one of the outputs high is known as the forward delay.

The last four transistors, M8-M11, perform a NOR operation. These transistors are used to generate the $DONE_{out}$ signal from the signal OUT and it's inverse. The NOR output is low whenever one or both of the inputs are high. As the ECDL circuit is enabled, one of the outputs will go high, which causes the NOR to go from high to low, resulting in a $DONE_{out}$ signal.

In the timing diagram of the handshaking sequence shown in Figure 10, the circuit is triggered by a high-to-low transition on the $Done_{in}$ signal (event a in Figure 10). The output of the cell is generated during the period when $Done_{in}$ is low. When the $Done_{in}$ signal goes high again (event d), the two outputs go high (event e) and the output value is invalid. This makes the ECDL circuit inherently four-phase with regards to clocking.

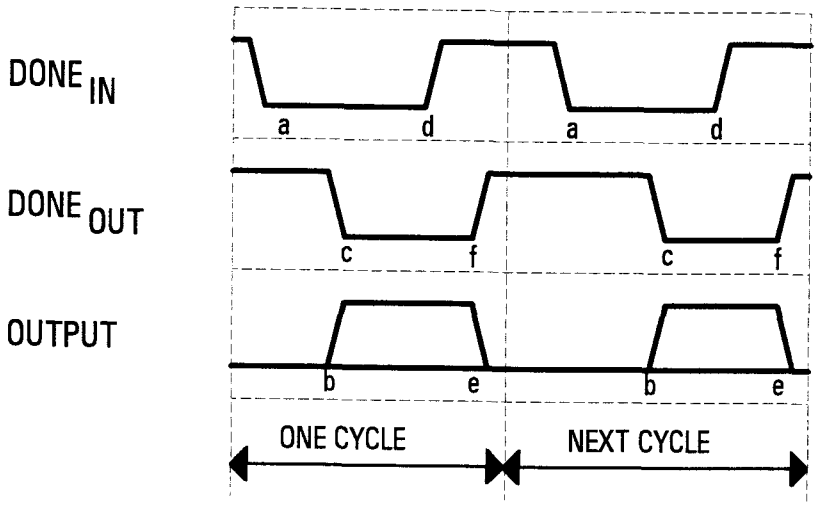

Figure 10. Timing Diagram for the Micropipeline.

The ECDL has a delay, from event a to events b and c, called the forward delay. This is the time required for the circuit to generate the outputs and the completion signal. The ECDL circuit also has a backward delay from event d to events e and f. This is the time required to reset both outputs low (event f). Both outputs must be low before the circuit can perform it's next cycle, since otherwise $Done_{out}$ will not go from low to high (event e). The backward delay time (or reset time) can add significantly to the delay of the cell, as will be seen when the simulation results from the multiplier are discussed in chapter 4. In the next section. we will discuss the generation of initial tokens.

3.3 Initial Token Generation

Having examined the ECDL circuit and the modified micropipeline based on it, we will now describe initial token generation. The initial tokens are generated internally. The "token" consists of putting a micropipeline stage in a different state than the state before or after it. With a token present, the Muller C-element will have its inputs at 00 or 11. If either an acknowledge comes from

the next stage or a request comes from the previous stage, the Muller C-element will trigger and begin a processing cycle in the current stage.

The initial token generation is done by inverting tokens across the boundaries between micropipeline stages. The token inversions can be done in two different ways. The first method is to put an inverter on the $Request_{in}$ and $Acknowledge_{in}$ lines. This causes the tokens to be inverted as they go between stages, and generates the initial tokens. This also introduces delays in the control circuitry as the signals have to pass through the inverter.

A better method is to use preset and reset of the control circuitry to set up adjacent stages where the initial token is required. One stage is preset, and the next or previous stage (or both, if two tokens are desired) is reset. The only circuit elements which must be affected in this manner are the Muller C-element and the SETDFF. This is because the SETDFF and the Muller C-element are the only elements that must be reset or preset to set up the circuit. Once the Muller C-element and the SETDFF are in a definite state, the state of the XNOR and the ECDL circuits are determined.

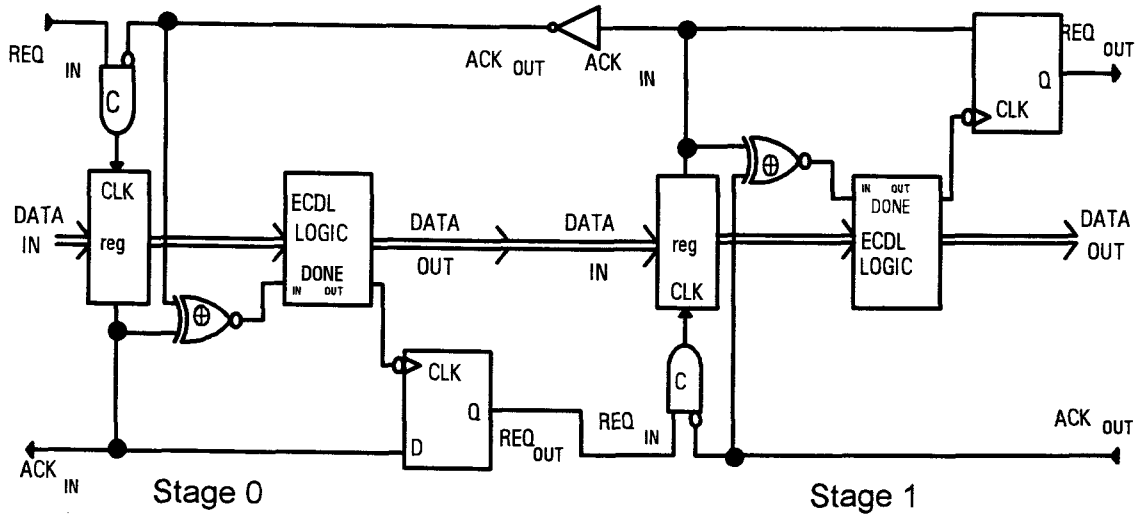

Figure 11. Two-stage ECDL pipeline.

In the following steps we will describe the generation of a token using the inverter method described above. It should be noted that a token exists when there is a request flowing through a pipeline, or when a request is waiting, such as at a join. Refer to Figure 11 as the following sequence of events is traced.

(1) The circuit is initialized with Muller C-elements = low, SETDFF = low, the ACK_{in} output of the second stage = low, and the ACK_{out} input of the first stage = high. The first stage Muller C-element is stable with any input to the REQ_{in}. The XNOR will output a low and trigger the ECDL circuit in the first stage. The REQ_{out} output from the first stage = low resulting in the REQ_{in} input to the second stage = high. Assuming a low on the second stage ACK_{out} input, the second stage Muller C-element is unstable.

(2) When the initialization signal is removed, the second stage Muller C-element = high.

(3) The second stage XNOR is triggered and its ECDL cell is enabled, causing an operation to be performed on input data. At the same time, the second stage ACK_{in} signal = high, and the ACK_{out} input of stage one = low.

(4) Assuming the REQ_{in} input is low, the Muller C-element in stage one remains stable. The XNOR inputs are now equal and it's output goes high. The ECDL circuit in stage one will reset.

(5) In the second stage, the D input of the SETDFF is now one. As the ECDL circuit completes it's task, it sends a high-to-low transition to the SETDFF, changing the output state Q from low to high.

(6) This change on the REQ_{out} line triggers the next stage, sending the token on down the micropipeline.

The chip was designed so that two or more of the chips could be placed in series which gives a FIR filter with more than three stages. To facilitate modularity for the FIR filter, all control signals are sent outside the chip. This also makes testing easier. More importantly, there had to be a Y_{in} and an X_{out} port. The Y_{in} port may not be used in a small one-chip filter, and it may not be used in the first chip of a multi-chip filter. This is because the Y_{in} is often zero for the first stage of a FIR filter. However, for multi-chip filters it must be included to carry through the results of previous chip stages. Similarly for X_{out}, we wouldn't normally need the delayed values of the X variable coming out of the last stage. However, the next chip in a multi-chip filter will need these delayed X values. The only other choice is to delay the X variable in a shift register outside the chip.

4. CIRCUIT DESIGN, SIMULATION, AND LAYOUT.

4.1 Overall organization of the Processing Elements

In this chapter the design, simulation, and layout of the various subcircuits of the FIR filter design are discussed. First the control section, the Muller C-Element, the XNOR, the DETDFF, and the SETDFF are discussed, followed by the arithmetic blocks that form the data path. In the last part of this chapter, the control and data paths are integrated in an examination of the basic filter stage.

4.2 The Control Unit

The control unit of the FIR filter stage has two functions. First, it controls the flow of tokens into and out of the filter stage. Second, it must enable and disable the ECDL processing circuit based on the availability of tokens to be processed. There are four very basic circuit blocks used in the FIR filter control unit design: the XNOR; Muller C-Element; the SETDFF; and the DETDFF. We begin with the Muller C-element.

4.2.1 The Muller C-element

The Muller C-element performs an AND function for tokens, synchronizing the current filter stage with the next and previous stages. In the micropipeline, the current filter stage cannot fire until two events have occurred: (1) The previous stage(s) must have completed their processing and output token(s) to the current stage. (2) The next stage(s) must have read in the token from the current stage, allowing the current stage to change it's outputs.

The Muller C-element is connected to the $Request_{in}$ signal from the previous filter stage, which signals when the previous stage has completed it's

processing. The other (inverted) input is connected to the $Acknowledge_{in}$ signal from the next filter stage, which acknowledges the receipt of the current stages' output token. These two connections satisfy the requirements for synchronizing token exchange between filter stages. Figure 12 (a) shows the logic symbol for the Muller C-element with one inverted input.

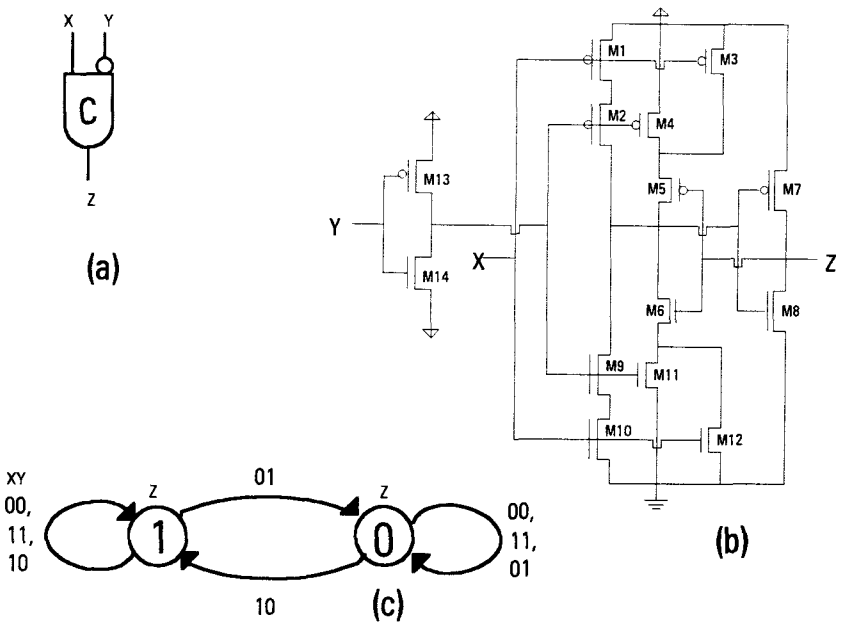

Figure 12. Muller C-Element logic symbol, circuit diagram, and state diagram.

As the state diagram in Figure 12 (c) shows, the Muller C-element has two states. Whenever the input X and the inverted input Y match, the output Z of the Muller C-element assumes the value of the X input. In all other cases, the output remains in it's current state. Table 1 shows the logic table for this circuit.

Table 1. Logic Table for the Muller C-element with 1 inverted input.

INPUT X	INPUT Y	CURRENT STATE	NEXT STATE
0	0	Q	Q
0	1	X	0
1	0	X	1
1	1	Q	Q

The following table shows which transistors are on and off with varying inputs to the circuit and varying current states.

Table 2. State Transition Table for the Muller C-element with one inverter.

X	Y	Z^{-1}	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	Z
0	0	0	ON	OFF	ON	OFF	ON	OFF	OFF	ON	ON	OFF	ON	OFF	ON	OFF	0
0	0	1	ON	OFF	ON	OFF	OFF	ON	ON	OFF	ON	OFF	ON	OFF	ON	OFF	1
0	1	X	ON	ON	ON	ON	ON	OFF	OFF	ON	OFF	OFF	OFF	OFF	OFF	ON	0
1	0	X	OFF	ON	OFF	ON	OFF	ON	ON	OFF	OFF	ON	OFF	ON	OFF	ON	1
1	1	0	OFF	OFF	OFF	OFF	ON	OFF	OFF	ON	ON	ON	ON	ON	ON	OFF	0
1	1	1	OFF	OFF	OFF	OFF	OFF	ON	ON	OFF	ON	ON	ON	ON	ON	OFF	1

In the circuit diagram in Figure 12 (c), some transistors are shown larger than others, since they are used to change the state of the C-element. The smaller transistors are only used to hold the value in it's current state, making the gate fully static.

4.2.2 The XNOR

The XNOR logic gate is implemented in standard CMOS logic, shown in Figure 13 [WE88]. The XOR circuit (M1-M6) consists of two inverters (M1-M2, M3-M4) and a pass transistor pair or transmission gate (M5-M6). A third inverter (M7-M8) is used to invert the A⊕B signal to get the A⊕Bbar output. Table 2 shows the state transitions for the XNOR circuit.

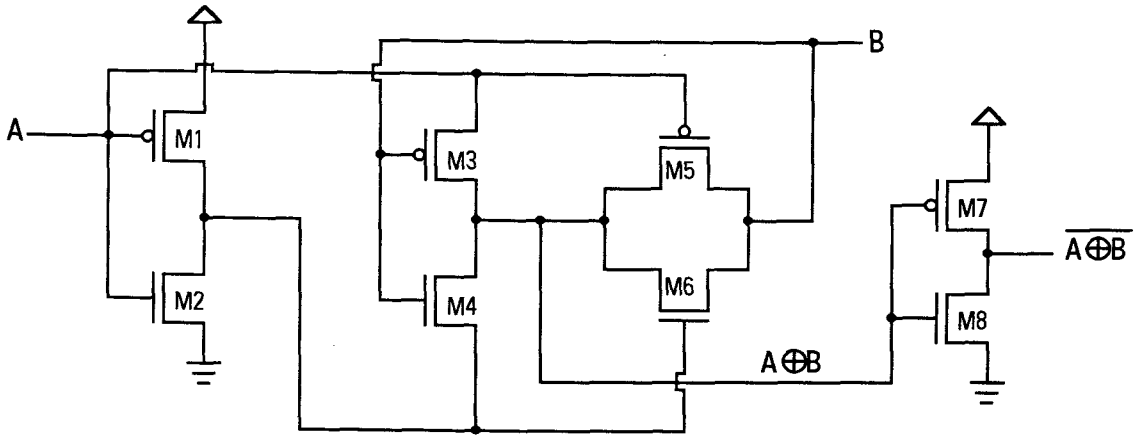

Figure 13. Basic XNOR circuit

Table 3. State Transition Table for the XNOR circuit.

A	B	M1	M2	M3	M4	M5	M6	M7	M8	$\overline{A \oplus B}$
0	0	ON	OFF	ON	OFF	OFF	OFF	ON	OFF	0
0	1	ON	OFF	OFF	ON	OFF	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	ON	ON	OFF	ON	1
1	1	OFF	ON	OFF	ON	ON	ON	ON	OFF	0

4.2.3 The Flip-Flops

The FIR filter stage uses a SETDFF and a DETDFF. The SETDFF is level triggered, and the DETDFF is edge triggered. The SETDFF is

implemented using pass transistors and NOR gates. The DETDFF uses an optimized design found in [LU90]. The SETDFF has both set and reset available, whereas the DETDFF does not. The SETDFF uses a 2-phase clock, requiring that both CLOCK and CLOCKbar be input. The DETDFF uses a single clock phase, but requires both D and Dbar. The logic symbol for the DETDFF is shown in Figure 14 (a).

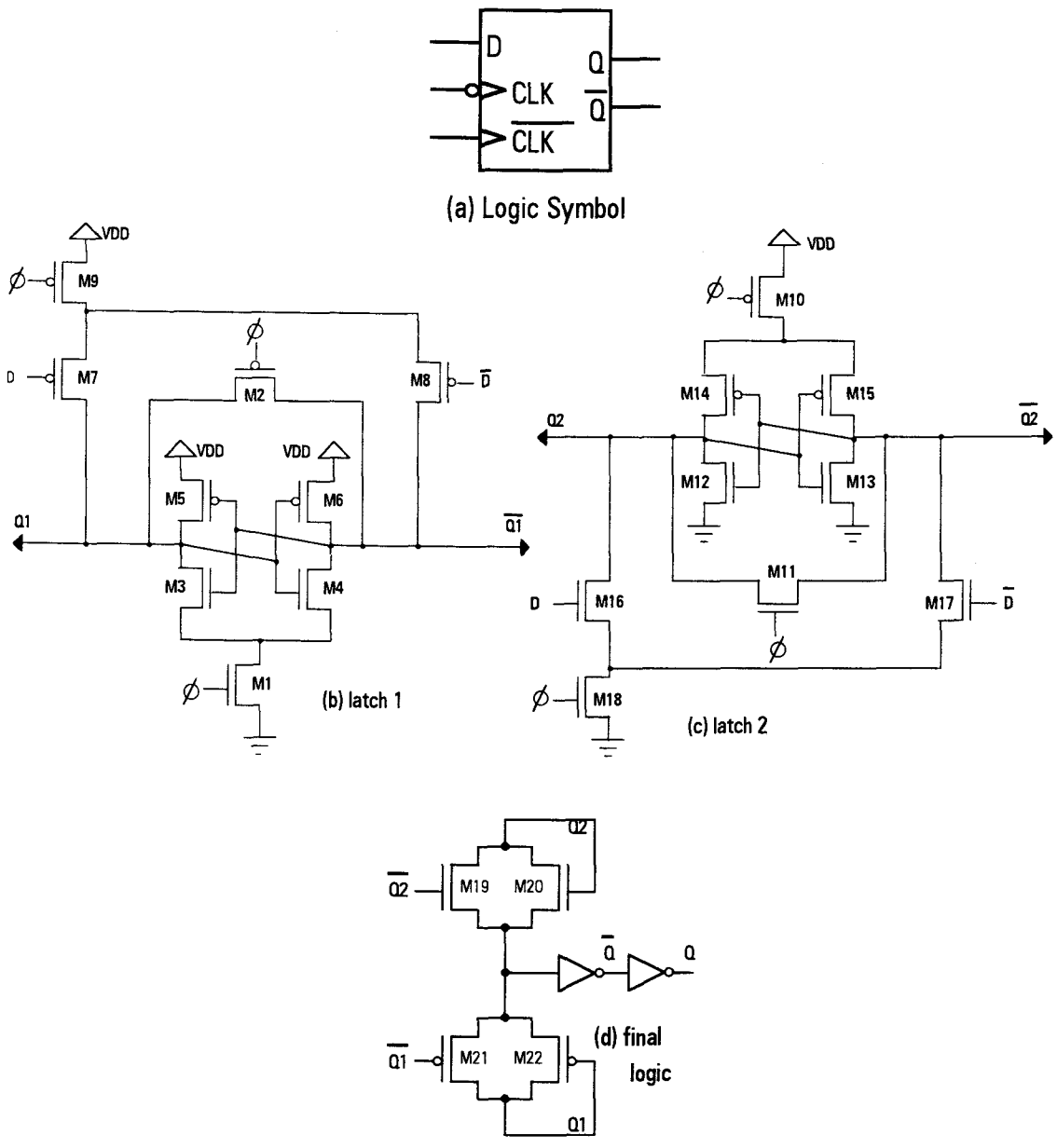

Figure 14. Double-Edge-Triggered D flip-flop Diagram.

The DETDFF circuit shown in Figure 14 has three main parts. The latch sections shown in Figure 14 (b) and (c) are used to generate partial results, one for each clock transition. Each latch is composed of a central flip-flop made up of two cross-coupled inverters (M3-M6 in Figure 14 (b), M12-M15 in Figure 14 (c)). The other transistors control the cross-coupled inverters. The latch in (b) is active during the low-to-high transition and the latch in (c) is active during the high-to-low transition. The third part of the circuit is the final logic, shown in Figure 14 (d). It takes the two values generated by the latches and combines them to produce the output Q which copies the input D during both clock transitions.

Table 4. State Transition Table for the DETDFF Latch 1 circuit.

D	∅	M1	M2	M3	M4	M5	M6	M7	M8	M9	Q1	$\overline{Q1}$
0	↓	OFF	ON	OFF	OFF	ON	ON	ON	OFF	ON	1	1
0	↑	ON	OFF	ON	OFF	OFF	ON	ON	OFF	OFF	0	1
1	↓	OFF	ON	OFF	OFF	ON	ON	OFF	ON	ON	1	1
1	↑	ON	OFF	OFF	ON	ON	OFF	OFF	ON	OFF	1	0

We will examine the operation of the latch 1 and final logic circuits of the DETDFF, since the latch 2 is a mirror image of latch 1 and operates in much the same way. The sequence of events as the DETDFF goes through one cycle is as follows:

(1) The two outputs in latch 1, Figure 14 (b), are held to VDD when the latch is off (when the clock is low). Transistor M9 is on, and either M7 or M8 will be conducting Vdd to one of the outputs, with a slight reduction due to V_t.

Transistor M2 will be conducting Vdd to the other, with another reduction in voltage due to V_t. In the final logic (d), with both inputs high (Q1 and Q1bar) the lower pass transistors M21-M22 are cut off.

(2) When the clock transitions high, the path to Vdd through M9 and the path shorting the two outputs through M2 are cut off. The transistor M1 is now on, which provides a pathway to ground for the inverters.

(3) With the inputs to both inverters high, both inverters (M3, M5) and (M4, M6) will be trying to bring their outputs low. The side that was already lower in voltage due to the V_t drop through M2 will be successful, and the other will remain at high.

(4) In the final logic, one of the transistors M21 or M22 will be on, bringing the output Q1 to the input of the series inverters. The output of the inverters will generate both Q and Qbar.

Latch 2 operates in the same way, except that the outputs are driven low instead of high in the off clock phase. One side will be driven high by the inverter pair. The DETDFFs are used to store the 4-bit token values in each filter stage. There are 16 of them in each filter stage, organized in four banks of four each.

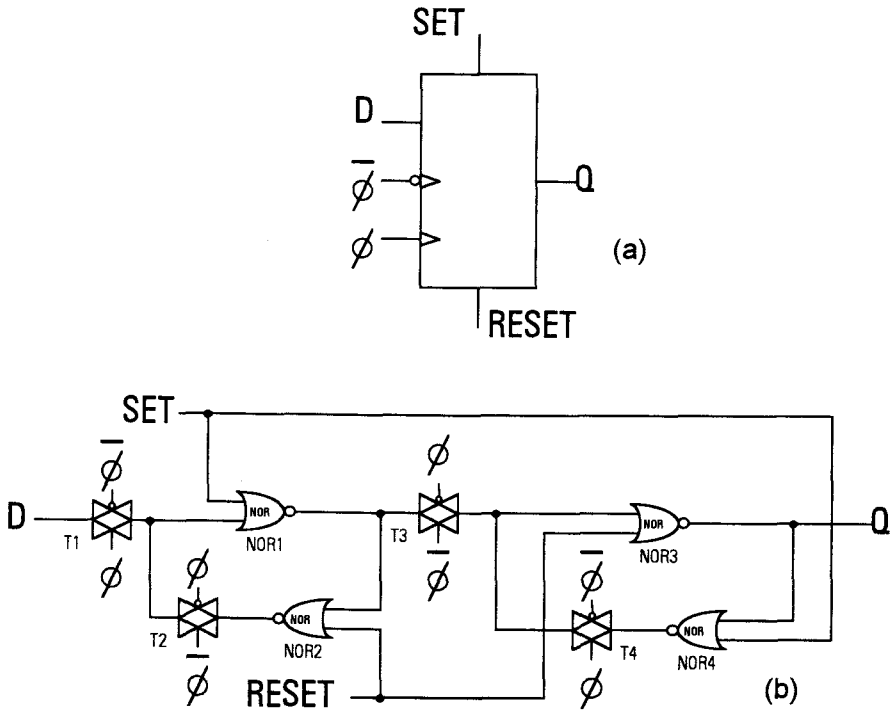

Figure 15. SETDFF (Single-Edge-Triggered D Flip-Flop).

The SETDFF in Figure 15 is a design found and described in [WE88]. It has both Set and Reset available, and is triggered on the falling edge. The NOR gates are only needed when Set and Reset are desired, or else the circuit could have used inverters instead. The pass transistors and NOR gates act as a two-stage flip-flop, holding the value and passing it to the output when the clock has made one complete low-high-low cycle. In Figure 15 (b) transmission gates (T1-T2) and NOR gates (NOR1-NOR2) are the first stage, while (T1-T2) and (NOR1-NOR2) are the second stage.

The circuit requires a two-phase clock, which leaves it open to clock skew problems. If the clocks were both positive or both negative at the same time, the input D could be connected to the output of NOR2 and the outputs of NOR1 and NOR4 could be shorted together.

The SETDFF circuit in Figure 15 (b) performs the following operations during each cycle: (1) The clock is high and input D is valid, so T1 is on and

input D determines the output of NOR1. T2 and T4 are off, but T3 is on and provides a feedback path for NOR3 and NOR4, maintaining the output in it's present state. (2) When the clock goes low, T1 turns off and D is disconnected. T2 is on and provides a feedback path for NOR1 and NOR2, maintaining the input D for the second stage. T3 provides a path from the output of the first stage to NOR3, setting the second stage output Q to reflect the input D.

4.2.4 Simulations

The simulation methodology for the Muller C-element was as follows: First the circuit parameters were extracted from the cell layout, and then the simulations were done using HSPICE for 2.0 micron technology. The simulations were performed on the control and data paths of one PE in a hierarchical manner. The simulation results for the control path circuits are summarized in Table 5 below.

Simulations revealed that the Muller C-element had an average delay of 3.5 ns. This is very high, and is due to transistors M1-M2 and M7-M10 being smaller than they should be. All of the other transistors in this layout are minimum size. Since it is the function of M1-M2 and M9-M10 in Figure 12 to override M5-M6, they should be much larger than M5-M6. In the cell layout they are only 1.5 times as large. Because the Muller C-element delay is in the critical path of the circuit, the larger-than-expected delay of the Muller C-element will cause the FIR filter stage to have lower throughput.

It was expected that there would be current spikes when the output of the Muller C-element changes, but simulation showed that there were also current spikes when the inputs are changed from $X \neq Y$ to $X = Y$. These spikes are caused by changes in the circuit feedback path, which holds the output steady. When the inputs differ, the output is held constant by the action of M7-M8 in Figure 12.

When the inputs are equal, the output is maintained by transistors M5-M6 in Figure 12.

Since the heart of the Muller C-element is the two cross-coupled inverters M5-M8, the power consumption is similar to that of an inverter. When the output is changing state, the peak power consumption is 2.8 mW. At 18 Mhz, the average power consumption is 0.2 mW. When the Muller C-element has static inputs, like an inverter, it consumes only leakage current. The Muller C-element has a total of 14 transistors. With 2.0 micron CMOS layout rules, it uses an area of 90 by 80 microns.

The XNOR has a peak power consumption of 3.6 mW at switching, and it's average power consumption is 0.06 mW at 18 Mhz. It's delay is 0.9 ns, and it's power-delay product is 0.1 picojoule. The cell layout takes an area of 40 by 80 microns.

The DETDFF uses a space of 70 by 145 microns when laid out in 2 micron CMOS. It has a peak power consumption of 3.0 mW, and an average power consumption of 0.1 mW at 18 Mhz. The delay time from clock transition to output Q changing is 1.9 ns. The other flip-flop, the SETDFF, uses an area of 115 by 90 microns, about the same as the DETDFF. It's peak power consumption is 2.0 mW, with an average power consumption of 0.1 mW at 18 Mhz. As it was laid out, a large amount of polysilicon layer was used. This contributed to the 1.6 ns delay.

Table 5. Summary of simulation results for the Control-Path circuits.

Circuit	Delay	Peak Power	Avg Power @ 18 MHz	Power-Delay Product	Number of Transistors	Area
Muller C	3.5 ns	2.8 mW	0.2 mW	0.7 pJ	7 P, 7 N	7200 μ^2
XNOR	3.6 ns	0.9 mW	0.06 mW	0.2 pJ	4 P, 4 N	3200 μ^2
SETDFF	1.6 ns	2.0 mW	0.1 mW	0.2 pJ	12 P, 12 N	10,350 μ^2
DETDFF	1.9 ns	3.0 mW	0.1 mW	0.2 pJ	13 P, 13 N	10,150 μ^2

4.3 Data Path

There were two different arithmetic circuit blocks used in the data path of each filter stage. They are the adder and the multiplier. Each uses unsigned integer values, and both are four bits wide. They were implemented in ECDL, and were developed and laid out by [LU91A]. The multiplier was the larger of the two, occupying a space of 475 by 750 microns. It is an array multiplier, with a four by five cell organization. The array organization works well with an ECDL circuit, since this lets calculation proceed as fast as the circuits will allow.

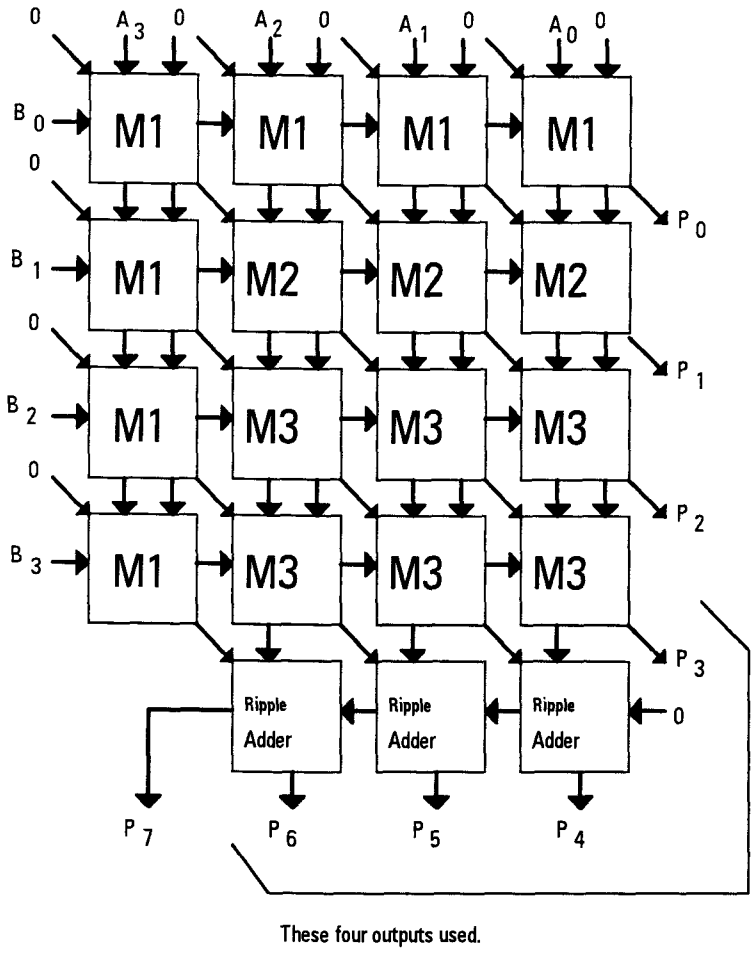

Figure 16. Multiplier Block Diagram.

4.3.1 Multiplier

The basic principle of operation of an array multiplier is the generation of partial products in parallel. There are two basic cell types in the array. There are the multiplier cells, M1-M3 in Figure 16, which are used to compute the partial products and low-nibble multiplier results. At the bottom of Figure 16 are the ripple carry adder cells, which are used to generate the final product. For the multiplier cells, there are several different types. The most general circuit, M3, is used in the center of the multiplier array. It is the most complex of all, since it must handle the most inputs. The other multiplier circuits, M1 and M2,

are simplified versions which can be used in the first and second rows and first column, where some of the inputs are always zeroes.

This multiplier takes two four-bit inputs, and generates an eight-bit output. There is no need for all eight of the bits, since the next stage in the pipeline will only take a four-bit input. Since the coefficient, which is one input to the multiplier, is assumed to be between zero and one, the output of the multiplier will have a binary point and a fractional part. If we let a coefficient input of 1000_2 be the "one" value, and 0000_2 be the "zero" value, then the binary point will be between bits 2 and 3. This can be seen from identity since 1000_2 times 1000_2 will give 01000.000_2 on the output of the multiplier. To preserve scaling of token values throughout the pipeline, only bits 3 through 6 should be connected to the next circuit.

Another way to organize the data path blocks is to assume that an coefficient input of 10000_2 would be 1. Such a 5-bit value could not actually be input, so a coefficient of 1111_2 would be the closest possible value to one. The binary point on the output would now be between bits 3 and 4, and bits 4-7 would be the output to the next filter stage. The advantage with this scheme is that the coefficients can be more accurate, since there are almost twice as many possible values for the coefficient. The disadvantage is that the filter cannot have coefficients of one, so it could not perform a simple delay function without attenuating the signal passing through it.

4.3.2 Adder

The other arithmetic circuit block is the adder. It is a four-bit carry ripple adder based again on ECDL [LU91B]. Adders of this type tend to be much slower than adders which try to predict what the carry bits will be to speed up the calculation. However, they have an advantage in that they are simpler, more

regular, and much smaller due to the lack of complicated carry lookahead logic. The block diagram in Figure 17 shows the organization of the ECDL adder.

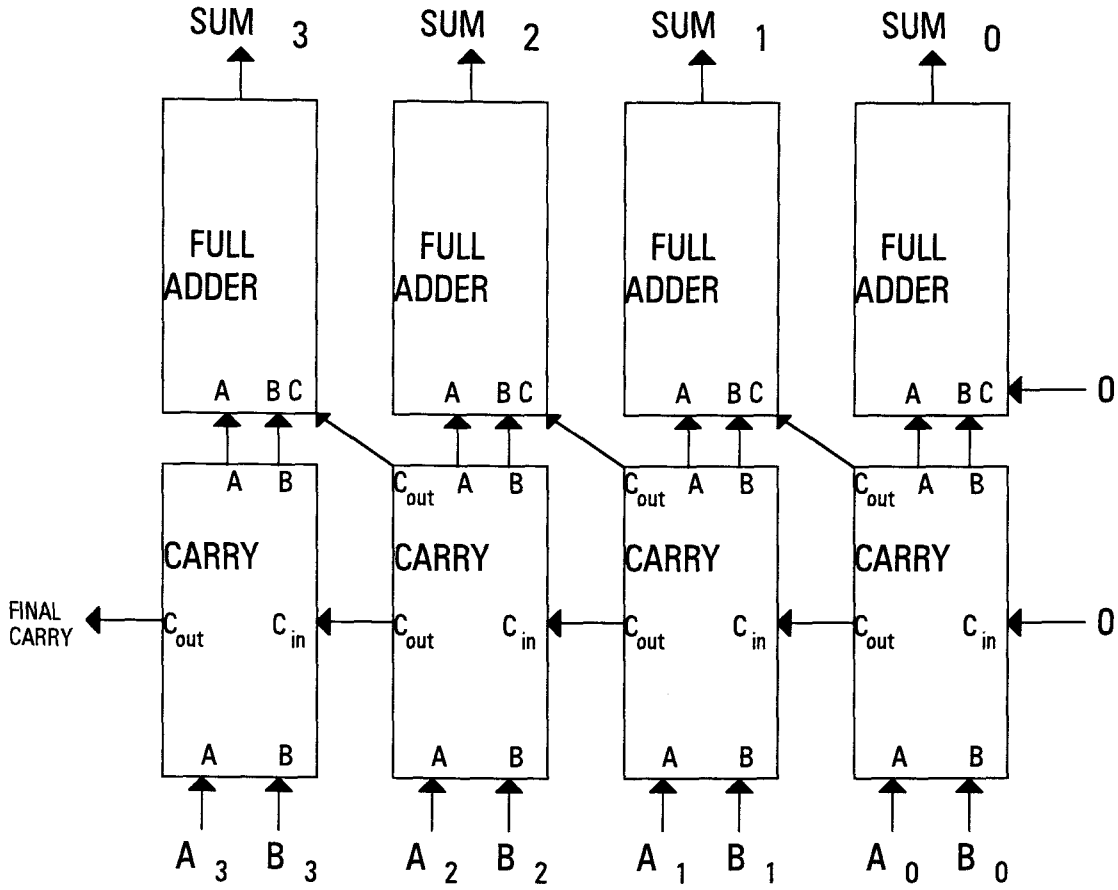

Figure 17. Adder Block Diagram.

An adder of this type implemented in ECDL has another advantage in that it will not produce false values during the carry ripple period. This is because the $Done_{in}$ control circuitry in the ECDL cell will not "fire" until all inputs are ready. By connecting the $Done_{in}$ of one stage to the $Done_{out}$ of the next lower bit carry subcell, the stage will not begin to produce a value until the previous stage has completed it's carry. Other circuit types such as static CMOS may produce false values on their outputs because the carry input to a stage may change during the setup of that stage. Static CMOS circuits continuously

evaluate their inputs and may produce temporary glitches before settling to the correct value.

4.3.3 Simulations

Simulation of the multiplier block indicated that there was a forward delay of about 15.6 ns. This is a reasonable value considering that it is implemented in 2 micron line width. To compare the simulation results from the multiplier with the results from the control circuits, consider that the longest path through the multiplier is 8 stages. This comes out to about 2.0 ns per stage. Interestingly, the backward delay was simulated at 19.75 ns, more than 4ns longer than the forward delay. This is probably caused by the fact that the transistors which discharge the output nodes of the ECDL gate are smaller than the transistors which make up the logic tree. The peak power consumption is 9.68 mA. This compares favorably with other circuit designs for a 4 bit multiplier. The simulation signal graphs for the control signals of the multiplier can be found in Figure 18 below. The inputs are not shown, but all other signals are shown and their names are given in Table 6 below.

Table 6. Signal names for the Multiplier simulation results.

Simulation subgraph number	Signal name
100	$Done_{in}$
101	$Done_{out}$
-I(Vdd)	Current for Vdd
31-37, 41-47	Output bits 1-4, 5-6

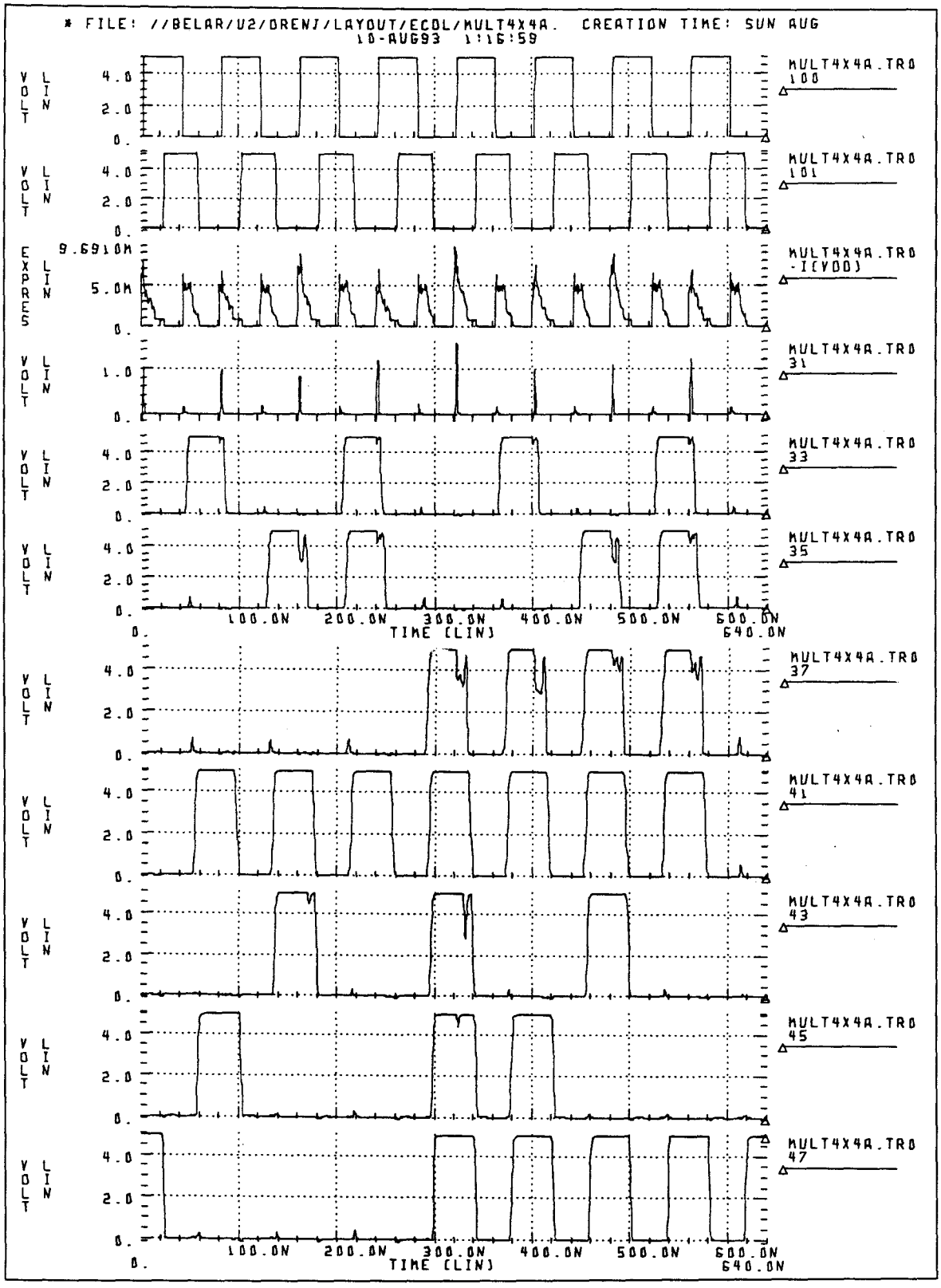

Figure 18. Multiplier Simulation Results

The adder block is smaller, and simulation of the adder indicated that it had a forward delay of only 7.4 ns. The backward delay was measured at 5.1 ns. The peak power was 2.8 mA, and the average power at 18 Mhz is 1.6 mW. Like the multiplier, this also compares favorably with results from other circuit types. Since the backward delay of the multiplier is so much longer than the forward delay of the adder, the adder will have long since completed it's cycle before the multiplier has the next input ready for the adder. This suggests that the multiplier is indeed the slowest section of the overall filter, and that it is critical to reduce the time taken by the multiplier to compute it's output. Signal graphs from the HSPICE simulation of the adder block can be found in Figures 19 and 20 below. The completed adder cell used a space of 220 x 420 microns.

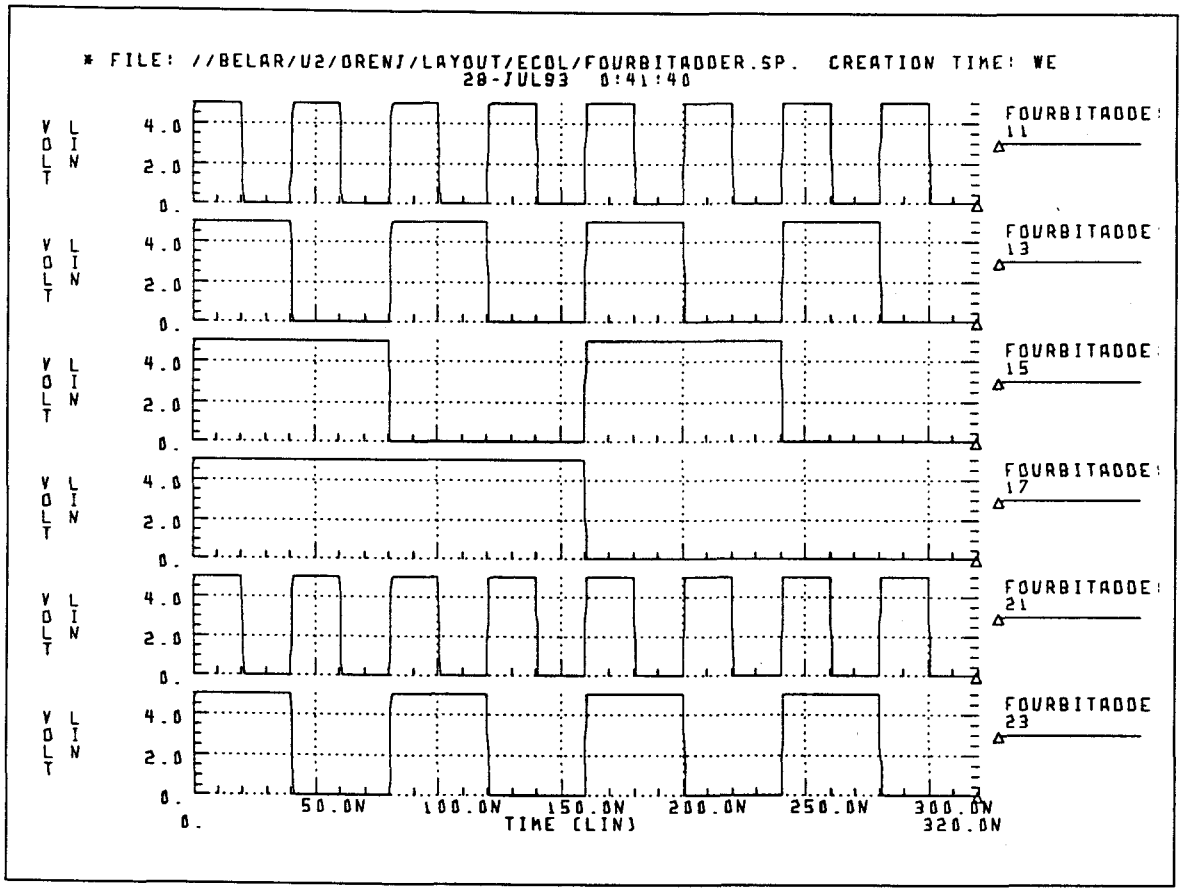

Figure 19. Adder Simulation Inputs

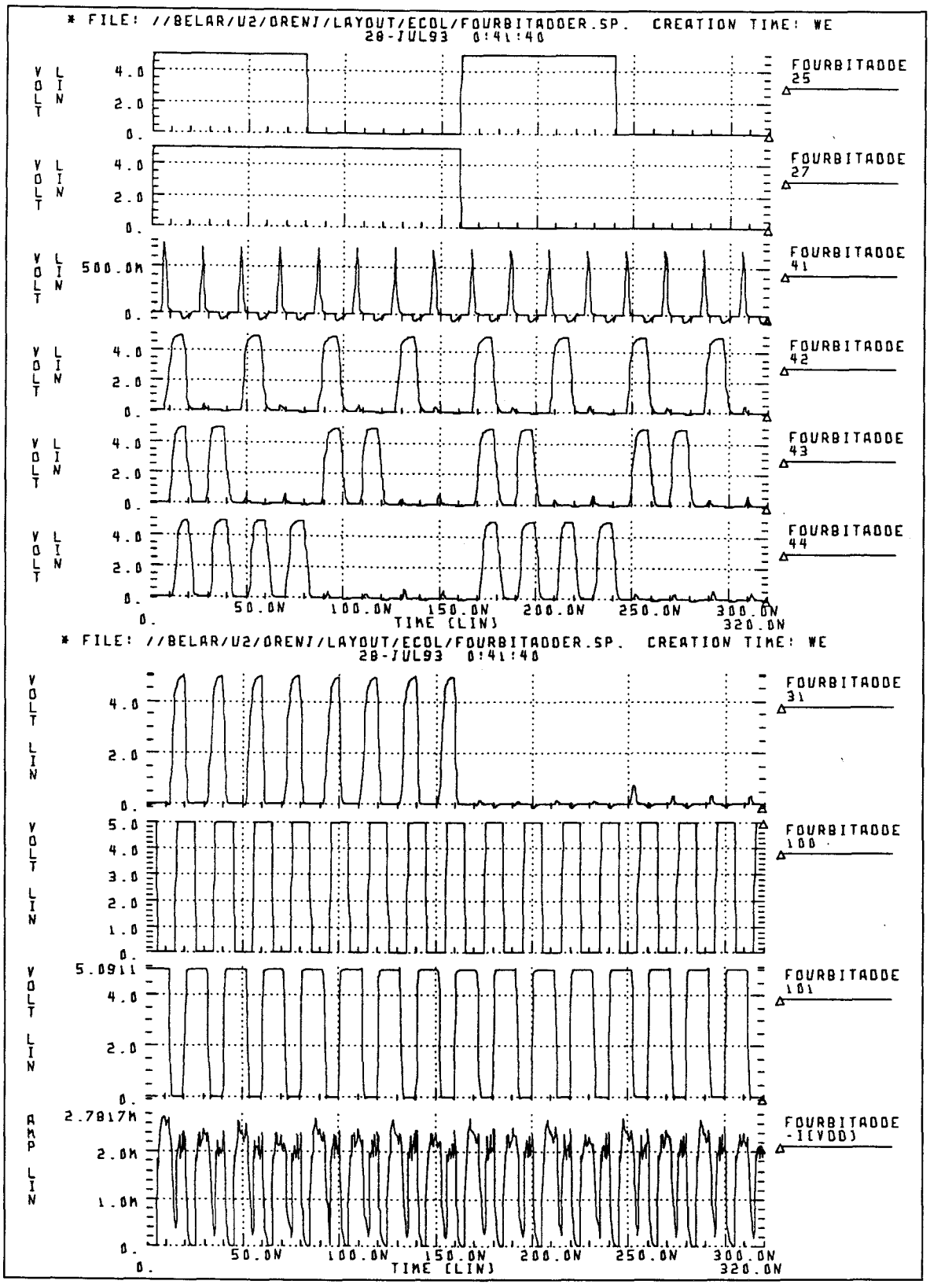

Figure 20. Adder Simulation Results

Table 7. Signal names for the Adder simulation results.

Simulation subgraph number	Signal name
11-17	Input A
21-27	Input B
41-44	Output bits 1-4
31	Carry output
100	$Done_{in}$
101	$Done_{out}$
-I(Vdd)	Current for Vdd

Table 8. Summary of simulation results for the Data-Path circuits.

Circuit	Forward Delay	Backward Delay	Peak Power	Avg Power @ 18 MHz	Power-Delay Product	Area
Multiplier	15.6 ns	19.75 ns	48.4 mW	5.7 mW	89 pJ	356,250 μ^2
Adder	7.4 ns	5.1 ns	14 mW	1.6 mW	12 pJ	92,400 μ^2

4.4 Overall Structure of the Filter

In this section I will combine all of the basic circuit blocks of the previous sections and discuss the layout of the main filter stages. I will begin by discussing the floor plan of the filter chip. Next I will present some of the layout issues, and discuss the simulation results for the filter stage cell.

4.4.1 Floor plan of the Filter stage

There are three major almost-identical cells on each chip. These cells, named triplet, are the individual three stages in the FIR filter. Each contains an adder and a multiplier, storage registers, and various control logic blocks. The

organization of these blocks is shown in Figure 21. This Figure shows the organization the same way as the plot of the actual fabricated chip in Figure 22.

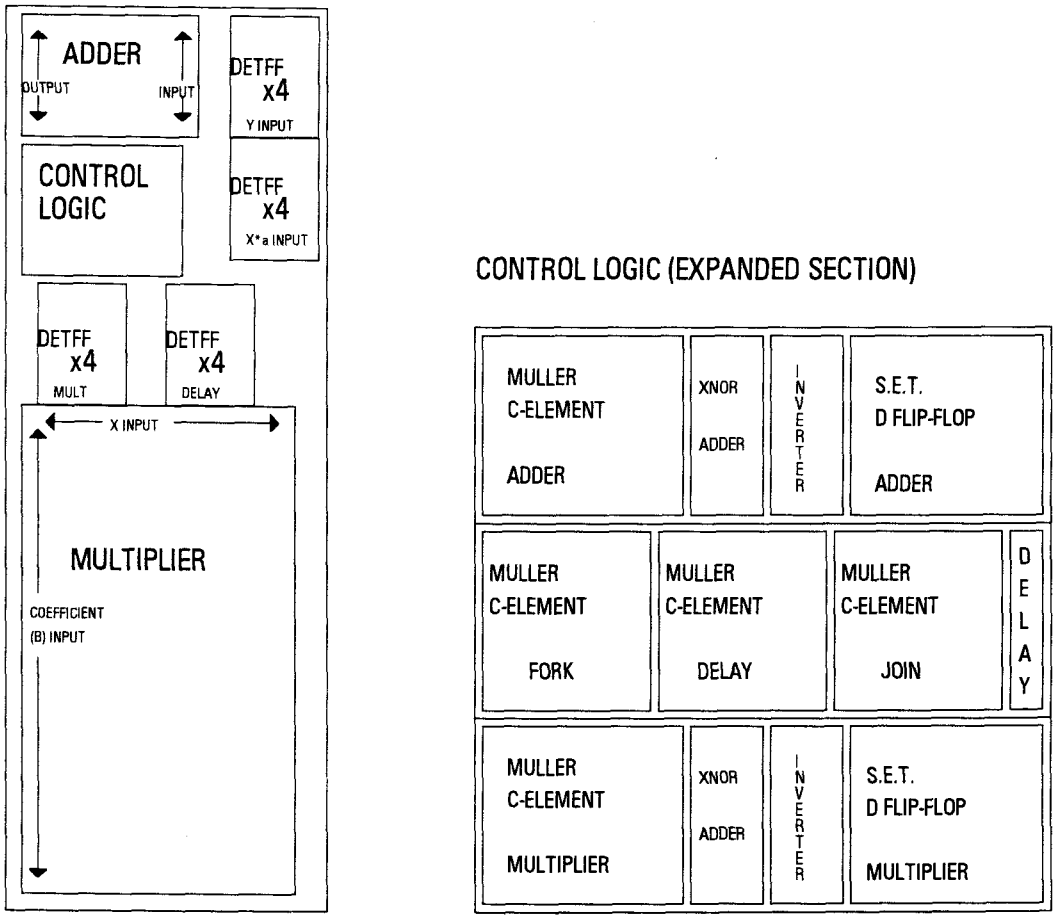

Figure 21. Floor diagram of the Triplet filter stage.

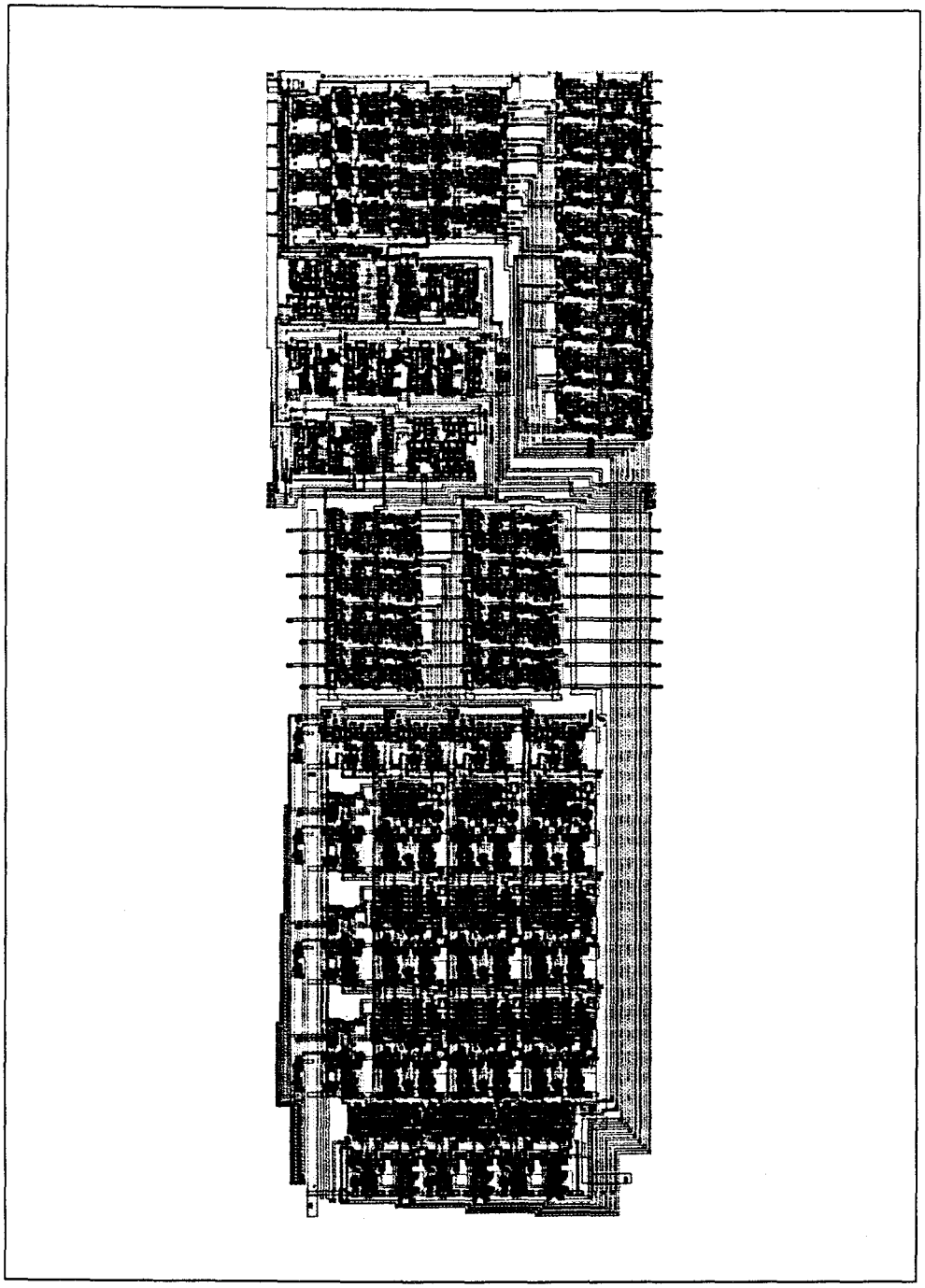

Figure 22. Plot of the filter chip.

Table 9 below summarizes the amount of chip area required for the Triplet cell. The multiplier was the largest cell by far, and uses up about 32 percent of the cell area. The DETDFF is the next largest area consumer, since

there are 16 of them. They used 15 percent of the area. The adder is third using about 8 percent of the cell area. The only thing that consumed more area than the multiplier was the interconnection wiring between all of the cells. The total area taken by the triplet cell is about 1700 microns long by 660 microns wide. Three of these triplet cells are placed side by side in the topmost level of the chip layout. With power busses, signal wiring, and space for IO pads and circuitry, the finished chip is 2300 by 2250 microns.

Table 9. Summary for the Triplet cell.

Cell Name	Number used	Total area used	Percent of area
Muller C	5	36,000 μ^2	3 %
XNOR	2	6400 μ^2	<1 %
DETDFF	16	162,400 μ^2	15 %
SETDFF	2	20,700 μ^2	2 %
Adder	1	92,400 μ^2	8 %
Multiplier	1	356,250 μ^2	32 %
Signal Interconnects	N/A	448,000 μ^2	40 %
Triplet Cell Total	N/A	1,122,000 μ^2	100 %

4.4.2 Layout issues for the filter stage

Alternate filter stages must be slightly different, since the control section of one must have both resets and presets, and the next must have resets only. In practice, all three filter stages had to be different due to interfacing with off-chip components. At one side of the pipeline, the X token must be input, requiring both the individual bits of the token and their inverse. On the same side, the Y value had to be output, requiring only the Y value's bits and not their

inverse. On the other side of the pipeline, the reverse was true, requiring a reversed wiring layout. The stage in the middle of the filter had to be different from the outside two to meet the requirement of alternating Preset/Reset stages.

The control circuitry for each filter stage was kept together in one single large block in order to facilitate wiring. The connections between the various control logic circuits would have required more area and would have been more difficult to route if they had to be routed across the filter stage cell. The centralized control block arrangement had the added benefit of allowing fast signal travel between the various control logic blocks. This signal timing benefit was not without it's perils. The short wires between the various blocks in one instance caused a timing error that was impossible to fix.

In the original micropipeline by Sutherland, the bundled convention for the control signals states that the control signals should be kept bundled with the data signals. This means that they should be routed the same way on-chip. The circuit is only guaranteed to operate properly when the delay of the Request signal is the same as the delay of the data signals.

Simulation revealed the difference in routing between control and data signals had caused a timing problem within each of the stages. The problem was caused by the Adder's $Done_{in}$ signal reaching the adder before the data to the adder was fully set up. This signal goes low as soon as the data have arrived and can be used. It is generated by the XNOR gate from the $Acknowledge_{in}$ signal and the output of the adders Muller C gate. The input data to the inverter that were not reaching the adder on time were from the multiplier, being generated on the opposite side of the single-stage cell. The signals had to travel across wires that were as long as 1mm, then through the DET D flip-flop at the input to the adder, and then to the input of the adder itself.

The control signal, by contrast, had to flow out of the multiplier, across a shorter path of approximately 0.5mm, and then through the logic blocks.

The path through the control logic consisted of the SETDFF, the $Acknowledge_{in}$ wire to the adder's control logic, and the XNOR gate. The control signal delay is shorter than the data signal delay by approximately 4 ns. This resulted in the lowest bit being incorrectly received approximately half of the time. Because the adder operates starting from the LSB and moving up to the MSB, only the accuracy of the LSB is affected. Since the adder performs the calculation of the carry bit first, and the sum bit second, the LSB carry bit is most likely to be incorrect. The upper input bits can tolerate extra delay because they are not needed until the lower bits have been computed.

We tried to fix the incorrect timing in the control block by placing delay inverters in the spaces available between the control logic cells and the adder cells. There was room for up to three inverters, but since signal inversion would have caused errors, only two inverters could be used giving a delay of about .0.8 ns. We needed to delay the signal at least 2 ns to guarantee accuracy in the LSB of the adder. To get more delay, we used a long diffusion run instead of a wire to connect the output of the delay inverters to the $DONE_{in}$ input of the adder. The large capacitance and large resistance of the diffusion run adds a delay of about 1.5 ns, delaying the control signal by about 2.3 ns. Timing was still marginal, and the circuit still exhibited errors in certain circumstances. When more than one bit is changing at one time in the output stream of the multiplier, the adder gives incorrect results in the lowest bit. These errors can be predicted easily in advance and can be factored into the results. The results must then be calculated on a computer and compared with the actual output of the chip. We will reexamine this timing problem when we show the simulation results.

4.4.3 Simulation results for the filter stage

The triplet cell was slow and difficult to simulate because it has 1300 transistors. The HSPICE program indicated that it had a peak power consumption of 45 mW, and average power consumption of 3.5 mW at 18 Mhz. The average cycle time or latency from $Request_{in}X$ to $Request_{out}Y$ was 37.8 ns. The graphs of the simulation results for the Triplet filter stage are in Figures 23 and 24. Table 10 shows the signals corresponding to the node numbers on the SPICE graphs.

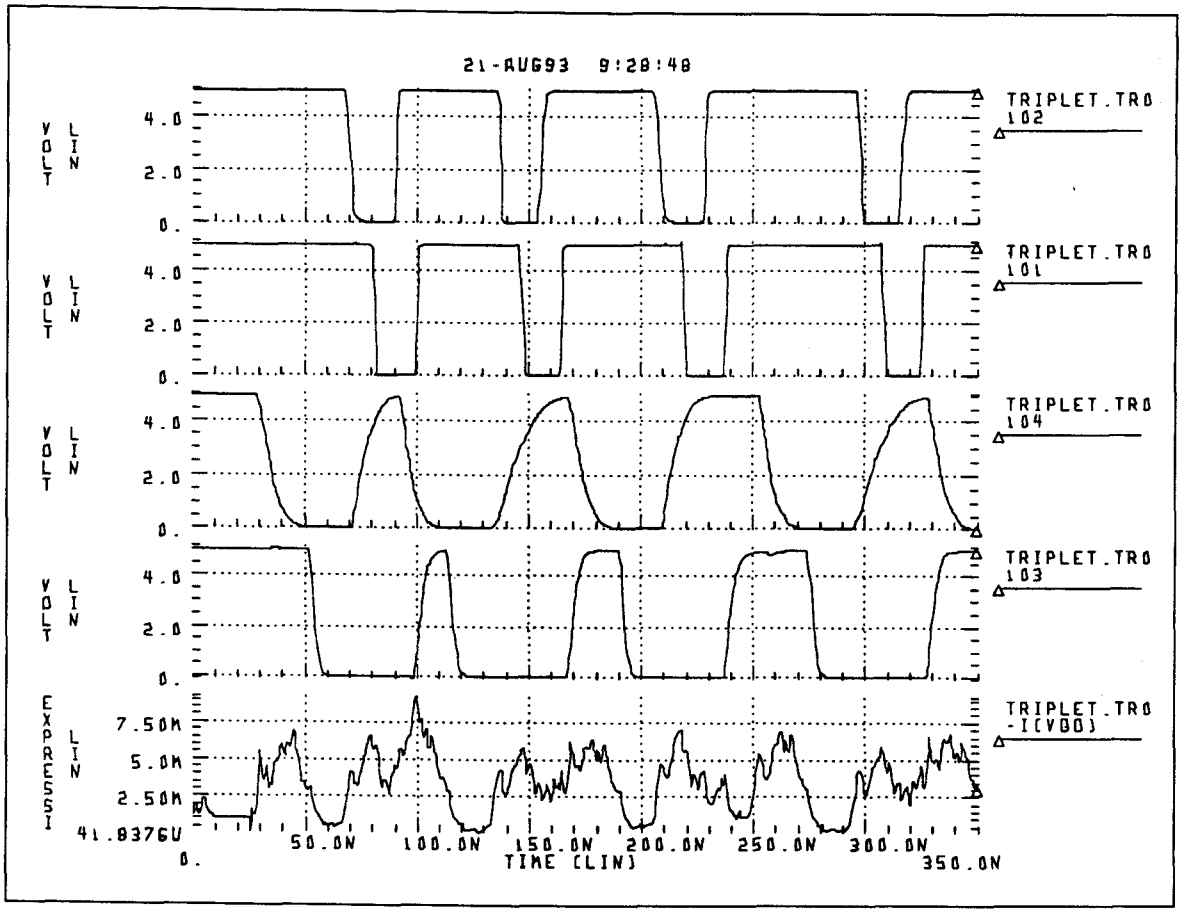

Figure 23. Triplet Simulation Results

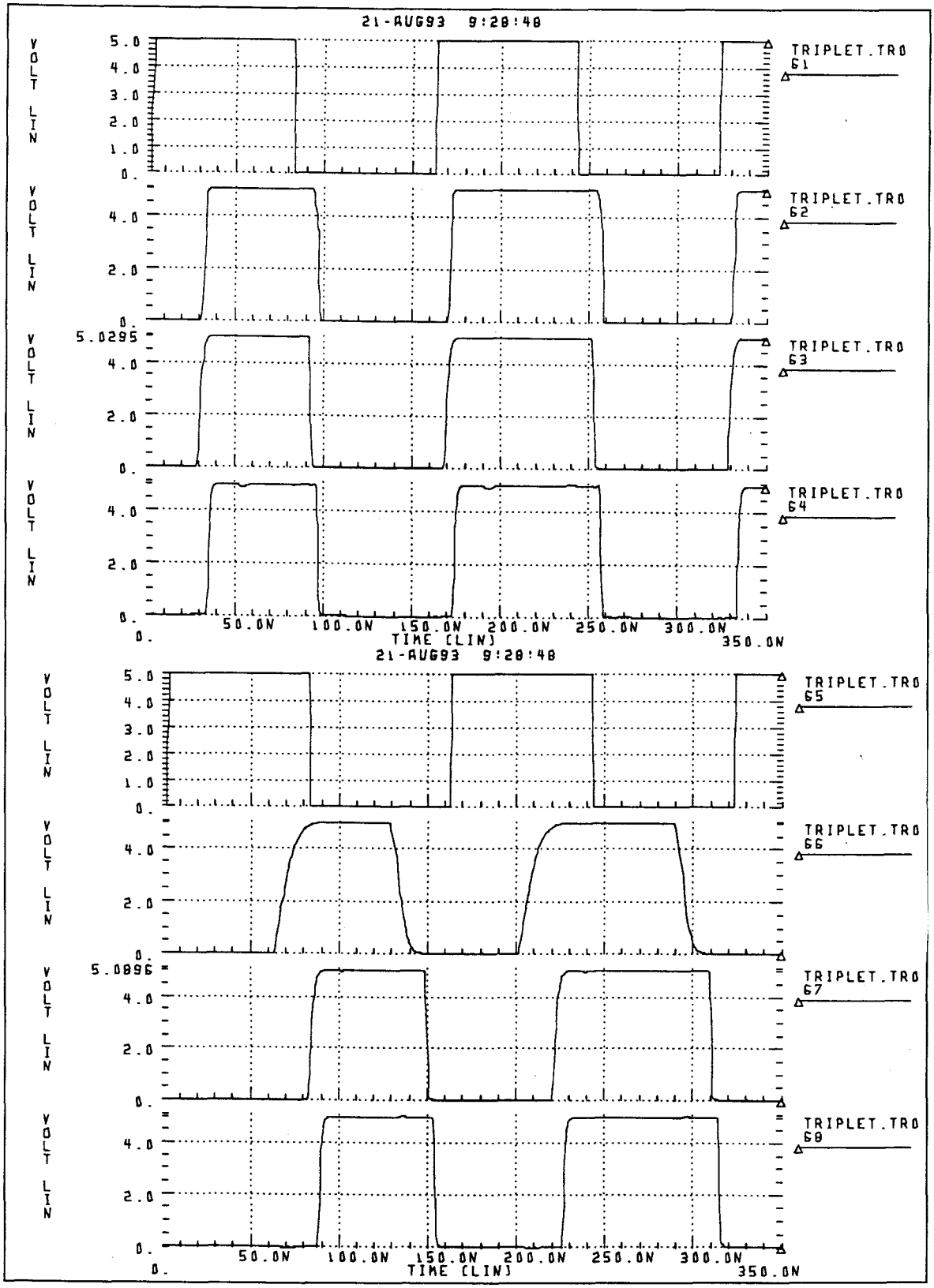

Figure 24. Triplet Simulation Results

Table 10. Signal names for the Triplet simulation results.

Signal subgraph number	Signal name
61	$REQ_{in}X$
62	$ACK_{in}X$
63	$REQ_{out}X$
64	$ACK_{out}X$
65	$REQ_{in}Y$
66	$ACK_{in}Y$
67	$REQ_{out}Y$
68	$ACK_{out}Y$
102	Adder $DONE_{in}$
101	Adder $DONE_{out}$
104	Multiplier $DONE_{in}$
103	Multiplier $DONE_{out}$
-I(Vdd)	Vdd current

For purposes of making the results of this research more easily comparable to other technologies, I have scaled the layout down to a 1.2 micron line width, and also scaled the results up to 8 bits at 2 micron. For 1.2 micron line width, the peak power consumption declines to 35 mW. The latency or delay time is reduced to 25.5 ns. When the design is scaled up to 8 bits, the delay or latency increases to 60.2 ns. The cycle time at 8 bits is 80.5 ns, giving a maximum speed of 12.4 MHz. The scaling from 2.0 μ to 1.2 μ and the scaling from 4 to 8 bits are summarized in Tables 11 and 12 below. In the next section I will discuss some aspects of the layout of the filter and the circuit blocks, and also the extraction and simulation of the circuit blocks.

Table 11. Comparison of 2μ and 1.2μ feature sizes for the Triplet filter cell.

Feature Size	Area	Power	Maximum Speed	Latency
2.0 micron	1.12E6 microns^2	45 mW	18 MHz	37.8 ns
1.2 micron	0.4E6 microns^2	35 mW	26 MHz	25.5 ns

Table 12. Comparison of 4- and 8-bit Triplet filter cells.

Number of Bits	Area	Power	Maximum Speed	Latency
4	1.12E6 microns^2	45 mW	18 MHz	37.8 ns
8	??E6 microns^2	??mW	12.4 MHz	60.2 ns

5. TESTING AND EXPERIMENTAL RESULTS

In presenting the results from the testing of this filter chip, I will begin by discussing the proper measurement of the output of the chip, and continue with a description of the critical path through the chip. I will then present the simulation results, starting with the smaller parts and working up to the filter stage as a whole. Lastly I will examine the results of the testing of the actual fabricated chip.

5.1 Measurement of Speed and Delay

Testing of this chip is made difficult by the fact that it is an asynchronous chip. With a synchronous design, we would only have to connect the chip to a clocked input circuit, and determine how fast we could clock the input before the filter failed. With an asynchronous chip, the input is assumed to come from a buffer. The simple input of the synchronous design cannot test the maximum capabilities of the filter, since the filter may take inputs faster or slower at various times. Nonetheless, the latency or delay time is well defined for both cases, and can be measured in a straightforward way.

We are interested in two different characteristics of the chip. The speed or cycle time is how fast the data can be clocked into the chip. The latency or delay time is how long it takes for a result to come out after a data value is input. The speed of the design in the synchronous case is well defined. It is the reciprocal of the fastest possible input period where the filter functions correctly. The asynchronous case is not so well defined. The speed must be measured over a period of time so as to get an average. This is because the speed, as well as the latency, may change drastically depending on the value of the inputs.

There is a distribution of speeds associated with the range of possible inputs values. The exact nature of this distribution will depend on the exact characteristics of the input source. This makes it necessary for the designer to look at the type of input values that are expected, and design the test cases accordingly. The larger the test suite, the more accurate the speed assessment will be.

The worst case value is usually used to get the speed of a design. This is so that the person designing with the chip can have a safety margin when designing a system around the part. This is the case for a synchronous design, where the input speed will not be changeable, and inputs that come too fast will produce incorrect results and failure. This cannot be done for an asynchronous design. For an asynchronous design, we want to look at the average case over the expected range of input values. The size of the input buffer will help determine the robustness of the overall design. A large input buffer will allow large swings in the type of input values, so that the actual performance of the design when looking from outside the filter and it's buffer will come closer to the average case.

As a practical matter, the chip must be tested in a synchronous manner. We have to use clocks to generate the control signals. To try to generate these signals in an asynchronous manner will cause too much clock skew and other problems, which will get worse as we get into the upper range of the filter. The best way to test a filter like this one is to put it into an asynchronous system. the next best way is to connect it to a synchronous test setup.

The critical path through the design will determine the speed of the filter. The critical path is the path from the input of values to the input of the next value. It is the path that needs to be evaluated most closely to get an accurate speed assessment. The worst case on this path will tell us something about the

performance of the filter. It will certainly tell us where the filter needs to be improved, and how large the safety margins are compared to the average case. But the average case through the critical path will give us a better indication of how fast the filter chip can be clocked.

In defining the critical path we must also define our assumptions about the current state of the filter chip. We will define the current state as having values in all registers, and tokens waiting at all of the joins. This is the state at rest after the chip has been operating for some time. Also, signals have already been presented (the $Request_{in}$ signals) for the values for the next operation. The operations critical for the calculation of the next output value will now be determined.

The critical path for the triplet filter stage is shown in Figure 25 below. Only the output stage is in the critical path through the filter. This can be seen since the next value to be output depends only on the previous stage's output value, and on the next X input value. The previous two stages can then be disregarded. The result of the previous stage's addition is waiting on the input to the last stages adder. The next output cycle begins when the $Request_{in}$ is presented to the X_{in} port. The X_{in} value itself is assumed to have already been set up. The $Request_{in}$ signal will cause the Muller C-element in the multiplier part of this filter stage to change state, after the Muller C-element delay. This will trigger the DETDFFs to bring in the X value, and also will trigger the XNOR gate. The XNOR gate will go low after a delay, triggering the ECDL multiplier. After a very long delay, the multiplier's $Done_{out}$ signal will go low, triggering the SET. D flip-flop. There will be a delay associated with that flip-flop, and then a delay associated with the Join Muller C-element. The output of the multiplier is complete at this point.

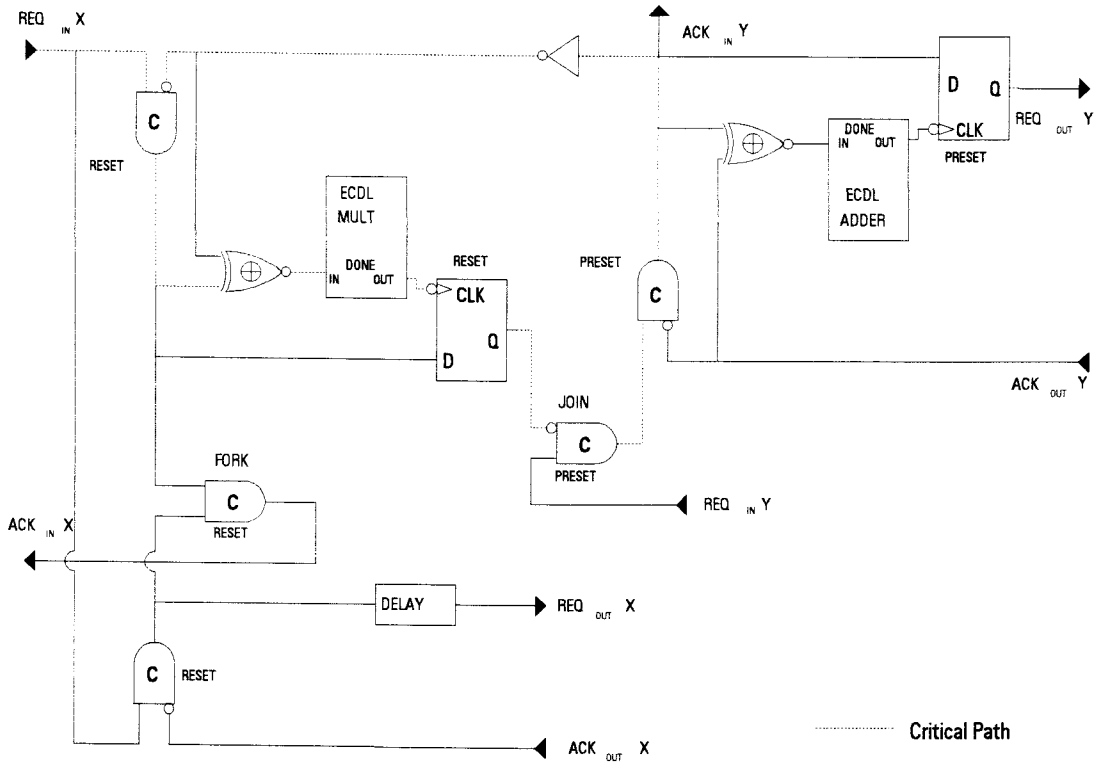

Figure 25. The critical path through the last stage.

The output of the Join Muller C-element will trigger the Muller C-element in the adder stage. After the delay, the adder's Muller C-element will change state, triggering the DET. D flip-flop, the XNOR gate in the adder stage, and the inverter on the Acknowledge line going back to the multiplier stage. The inverter, after a very short delay, resets the multiplier's Muller C-element. After a delay of about 4 ns, the multiplier's Muller C-element changes state, triggering the multiplier's XNOR gate to change back to the one state. This starts the reset cycle of the multiplier ECDL circuit, the longest delay in the whole cycle. At this point, the path diverges and we have separate cycle and delay paths.

Only after the multiplier is properly reset can the next X data value be input. If the next request comes too soon, the Muller C-element and the XNOR will change state properly, but the ECDL multiplier circuit will not have gotten a

chance to go high. Since it did not return high, the SETDFF will not function, and the circuit will be locked up. This is the backward delay of the circuit, and it is important to realize that it affects the maximum cycle rate at which data values can be presented. The backward delay in this case is long that the adder will have completed it's work before the multiplier starts on a new data value. In this way, the pipelining of the stages is of limited value internally, except in terms of delay for a given data value. The complete cycle time contains the backward delay of the multiplier, minus the delay of the Muller C-element and the XNOR circuit. Table 13 shows the calculations for the maximum time required to complete a cycle.

Table 13. Maximum cycle time for the filter stage.

Critical path element	Delay
Muller C-element,	3.5 ns
XNOR gate	0.9 ns
ECDL Multiplier, forward delay SETDFF	15.6 ns 1.6 ns
Muller C-element; Join	3.5 ns
Muller C-element; Adder	3.5 ns
Inverter	0.4 ns
XNOR gate	0.9 ns
ECDL Multiplier, backward delay Muller C-element,	19.7 ns -3.5 ns
XNOR gate	-0.9 ns
TOTAL	45.2 ns

At the same time the inverter between the adder and multiplier is changing state, the XNOR output is going low, enabling the ECDL adder. After the adder's delay the ECDL adder's $Done_{out}$ signal will go low, clocking the output of the adder's Muller C-element into the SETDFF. This will cause the

$Request_{out}Y$ signal to change state, informing the next circuit or buffer that another output value has been computed and is ready.

The actual time to complete the cycle is slightly longer, according to simulations. The simulations done from extract files on the triplet cell indicated that it could not be clocked faster than 56 ns with a synchronous clock. The reasons that it takes so much longer than the sum of the times from the individual cells are due to capacitance and resistance effects of the interconnections. There are many long runs of metal1 and metal2 layers connecting the various cells, and each has an associated resistance and capacitance. These resistances and capacitances slow down the output transistors in the cells driving these lines. These effects add up over many cells to make a difference of approximately 16 ns in the overall performance of the filter cell.

5.2 Test Results

In this section I will summarize the results from the testing of the chip. We received four copies of the chip, fabricated through MOSIS. All of the characteristics listed here are averages of the four chips. The average latency was 70.5 ns. The average power consumption was 36 mW at 4 MHz, including the power consumed by the I/O pads. The maximum speed could not be measured because it was necessary to divide down the clock generator to provide the clock phases needed. The maximum speed of the fabricated chip would be slower than the maximum speed in simulation due to the same I/O pad delays that slow down the latency path. The chip performed very well compared to expectations, with all values comparable to the values predicted by the HSPICE simulations.

In Figure 26 below, we will show some of the results from the testing. The first graph is the output of a state analyzer, connected up to one filter chip. The graph shows the inputs and outputs at the four ports of the filter chip. The control signals ($ACK_{in}X$, $REQ_{in}X$, etc.) cannot be shown on this graph, because they are to fast for the state analyzer to record. The X inputs to the filter is a ramp, starting at one at initialization, and proceeding to count up to 15. The coefficients are 1, 1/2, and 1/4. The Y_{out} port is seen to count up to 15 twice during the input sequence, which is correct.

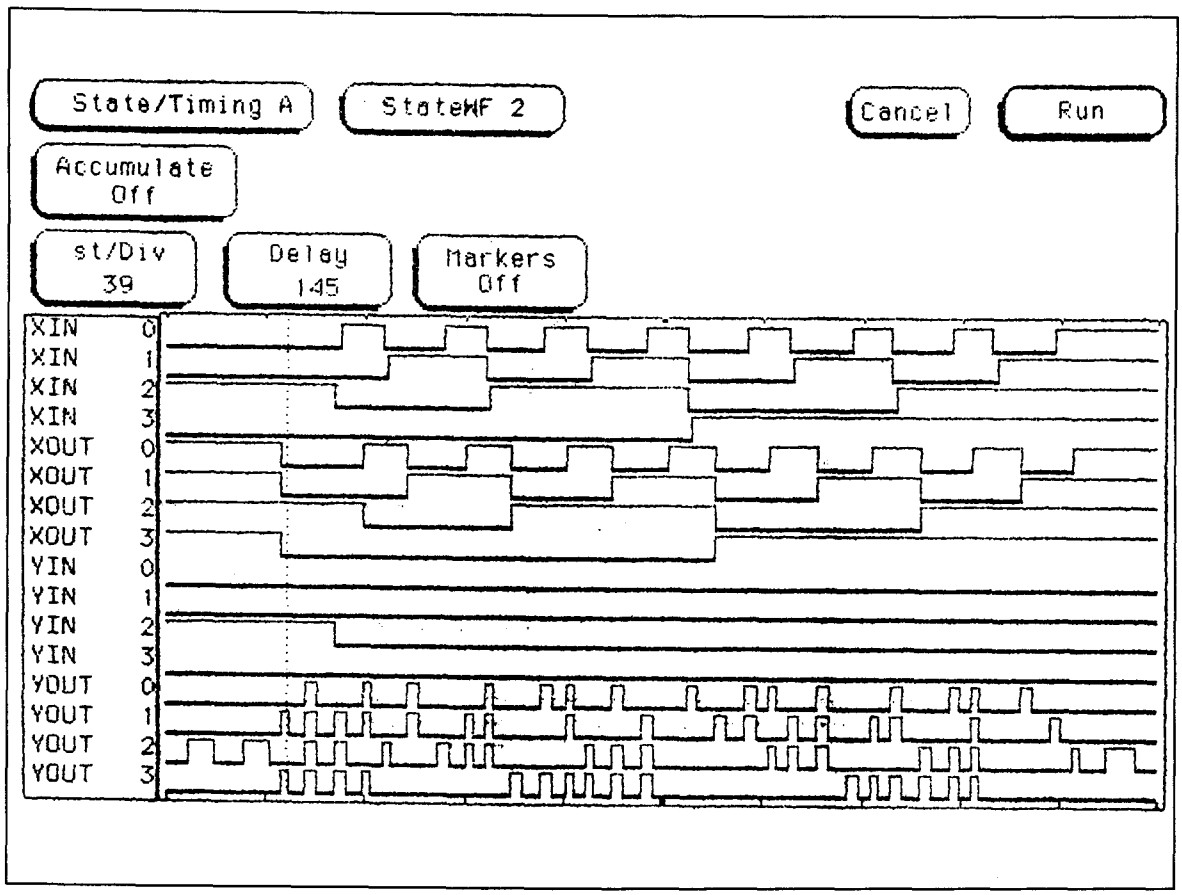

Figure 26. State analyzer graph for the filter chip.

An oscilloscope plot of the filter output pins is shown below in Figure 27. The two signals shown are $REQ_{out}X$ and $ACK_{in}Y$. Both of these signals are generated by the filter chip, in response to externally generated signals $REQ_{in}X$ and $REQ_{in}Y$. From this oscilloscope plot, the signals appear to be very noisy. This is mostly due to the noise generated by the oscilloscope probe.

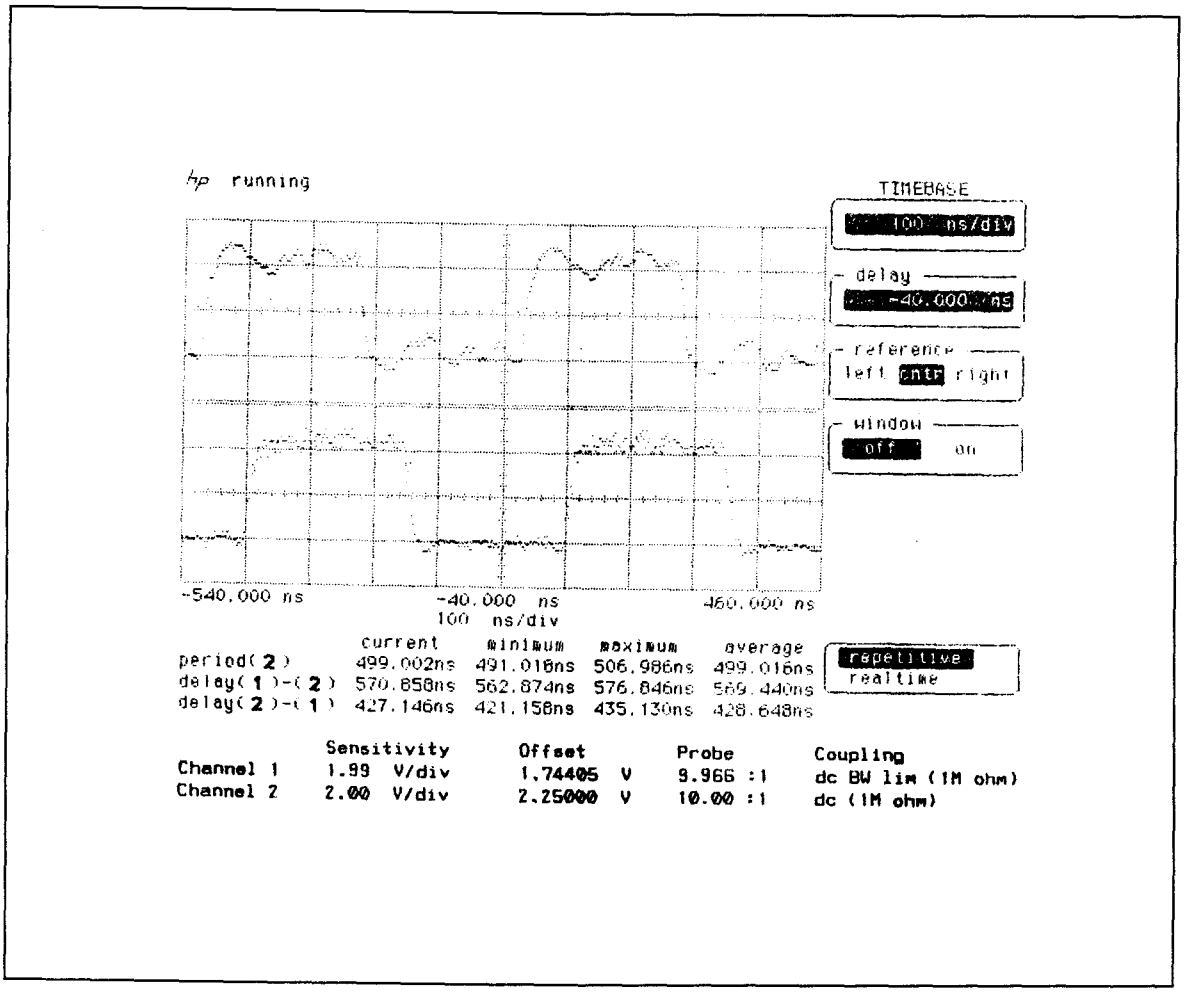

Figure 27. Oscilloscope plot for the filter chip.

6. CONCLUSION

A third-order self-timed filter has been produced. This filter uses four-bit coefficients, and has been produced for demonstration purposes. It uses the micropipeline control system developed by Sutherland, and the ECDL logic family developed by Lu.

Simulation indicates that the filter will work dependably up to about 18 MHz, with a 56 ns cycle time and a 37.8 ns latency. The simulated power consumption was 3.5 mW average, at 18 MHz. Experiments with the chip produced through MOSIS indicate a power consumption of about 36 mW from a 5V power supply, running at 4 MHz. The majority of this power was used by the I/O pin drivers. The measured delay time or latency from the time a data token is presented to when the result is output is 70.5 ns, including delays due to chip I/O pads. Simulations indicated that the same chip laid out in 1.2 micron line width would have a delay time of 25.5 ns, not including delays due to chip I/O pads.

One disadvantage in this filter architecture is that the backward delay of the ECDL multiplier is so large that the adder has long since finished it's calculation before the multiplier is ready to receive another input data token. This limits the effectiveness of the two-phase pipelined filter stage. The chip has advantages in speed and power consumption when compared to other designs.

Improvement of the multiplier should be the main focus of future research. It may be necessary to use a different coding scheme such as Booth encoding to speed up calculation of the results. It would be possible to use more than one multiplier, multiplexed, per stage, or alternatively to use one adder with more than one stage and multiplier to save on chip area. It might be possible to

improve the backward delay of the multiplier through an improved circuit. For a practical design, two's complement arithmetic must be used to allow both positive and negative data values without distortion.

BIBLIOGRAPHY

[AR86] Arvind and D. E. Culler, "Dataflow Architectures," Annual Reviews in Computer Science 1986, vol. 1, pp. 225-253, Annual Reviews Inc., 1986.

[JA88] G. M. Jacobs and R. W. Broderson, "Self-timed integrated circuits for digital signal processing applications," VLSI Signal Processing III, pp. 197-208, IEEE Press, 1988.

[LA88] C. H. Lau, D. Renshaw, and J. Mavor, "Data Flow approach to self-timed logic in VLSI," Proc. ISCAS, 1988, pp. 479-482.

[LU91] S. Lu and M. D. Ercegovac, "Evaluation of Two-Summand Adders Implemented in ECDL CMOS Differential Logic," IEEE J. Solid-State Circuits, vol. 26, no. 8, pp. 1152-1160, Aug. 1991.

[LU88] S. Lu, "Implementation of Iterative Networks with CMOS Differential Logic," IEEE J. Solid-State Circuits, vol. 23, no. 4, pp. 1013-1017, Aug. 1988.

[LU90] S. Lu and M. D. Ercegovac, "A Novel CMOS Implementation of Double-Edge-Triggered flip-flops," IEEE J. Solid-State Circuits, vol. 25, no. 4, pp. 1008-1010, Aug. 1990.

[LU88] S. Lu, "A safe single-phase clocking scheme for CMOS circuits," IEEE J. Solid-State Circuits, vol. 23, no. 1, pp. 280-283, Feb. 1988.

[LU93] S. Lu, "Design of Hardware Efficient Self Timed Circuits," Electronics Letters, vol. 29, no. 1, pp. 6-7, 7 January 1993.

[LU88] S. Lu, "Implementation of Micropipelines with ECDL," accepted for publication, IEEE Transactions in VLSI Systems, September 1994.

[QU91] P. Quinton and Y. Robert, "Systolic Algorithms and Architectures," Prentice-Hall, 1991.

[RO86] B. Rorabaugh, "Signal Processing Design Techniques," TAB Books, 1986.

[SU89] I. E. Sutherland, "Micropipelines," CACM, vol. 32, no. 6, pp. 720-738, June 1989

[UN81] S. H. Unger, "Double-Edge-Triggered Flip-Flops," IEEE Trans. Computers, vol. C-30, no. 6, pp. 447-451, June 1981.

[WE88] N. Weste and K. Eshraghian, "Principles of CMOS VLSI Design: A Systems Perspective," Addison-Wesley, 1988.